Hayden Donofrio
December 4 2020

Project Report - CS5350

Computing Environment

My computing environment consists of a 2015 13 inch Macbook Pro. Not the strongest environment there is! Going into the detailed specs, my machine consists of a 2.7 GHz Dual-Core Intel Core i5 processor. This processor has a turbo boost up to 3.1 GHz. Note that this is just an upper bound for the boost speed and it is not guaranteed that my processor runs at this speed. This processor belongs to the "Broadwell" Intel chipset. The processor has 3 MB shared L3 cache.

In regards to Memory this machine holds 8gm of 1766 MHz LPDDR3 memory. For storage the machine also holds 256 GB 2.7 GHz access time PCIe based flash storage. This machine has no graphics support and uses Intel onboard graphics (Intel Iris Graphics 6100).

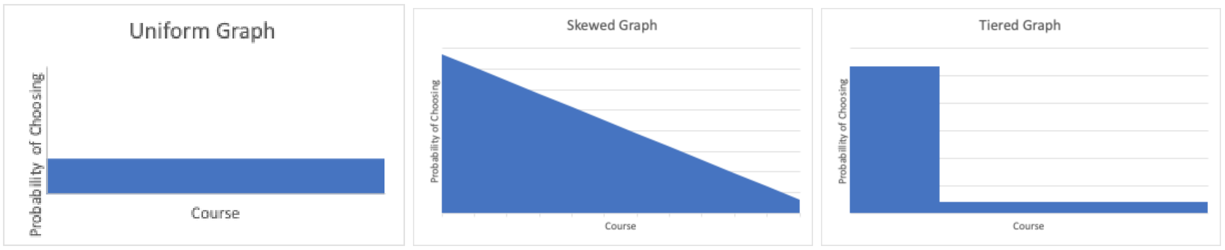
The OS is macOS Catalina 10.157.17. I am using the C++ language. Particularly, I am using CMake version 3.18.12. My compiler is Clang version 4.0.1. I am using the C++ 14 standard.

All statistics were all run on a previously idle system. I attempted to ensure that my program is the only large application running. However, it is not guaranteed that there is no interference from core system functions. We will discuss this throughout the rest of the report.

Generating Conflict Graphs

For my graph generation methods I have the following methods: Complete Graph, Cycle Graph, Random Graph E = 0, Random Graph E = V, Random Graph E = 25% density, Random Graph E = 50% density, Random Graph E = 75% density, and E = 100% Density.

The random graphs are generated in three different methods. The first method is a uniform random distribution. The second method of random graph distribution is the skewed distribution, this distribution makes it more likely to pick a vertex with a lower index number (1, 2, 3 being most likely). The final distribution is a tiered distribution, in my distribution the first quarter of all vertices receives 80% probability divided equally among each vertex. The other vertices receive the leftover 20% equally distributed among each vertex.

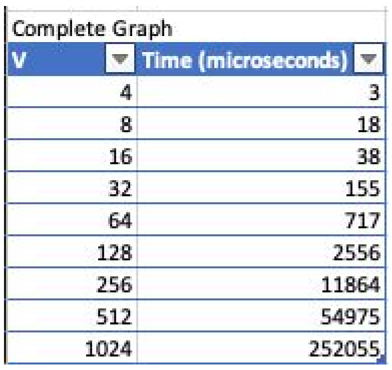


(Uniform, Skewed, and Tiered for **random graphs** probability for each vertex to be chosen)

Now we will look at the comparisons for each graph and gather analysis of them. We will go through the densities that I mentioned earlier for each random graph (E = {0,V, 25%, 40%, 75%, 100%}). The method for random graph generation is as follows: first the vertices are assembled in an array and multiplied by their probability of appearing. For example, if we have 12 vertices then the tiered random graph would have (13 * 12 / 2) entries. We would then associate entries to this, so the first 3 vertices would each have ((13 * 12 / 2) * .8) / 3 entries in the array and the other 9 vertices would each have ((13 * 12 / 2) * .2) / 9 entries in the array. I then shuffle the array in linear time by swapping each array with a random element selected uniformly in the array. Once that is done we generate a uniform random number in our set to select from our distribution array. It would have been much easier to generate all sets of pairs, but this technique did not provide very good “randomness” because the array would not follow the distribution after picking an edge. We wanted better randomness, so I decided to add the last step that generates a random number to index the array. If one edge is found to be a duplicate then we generate a different random edge and attempt to connect that edge to the graph.

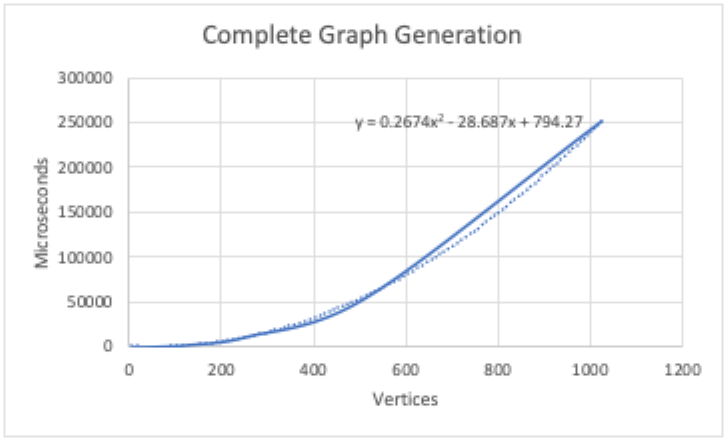
Now let's look at the time for all of our graphs and their conflict numbers. We will not be looking at conflicts for complete and cycle graphs because all conflicts are the same for all complete and cycle graphs. For all graph generation methods I averaged each time after 5 different generations, so each iteration would generate a different random graph with the same number of conflicts, not the same random graph 5 times.

Complete Graph Generation



Complete Graph

V	Time (microseconds)
4	3
8	18
16	38
32	155
64	717
128	2556
256	11864
512	54975
1024	252055

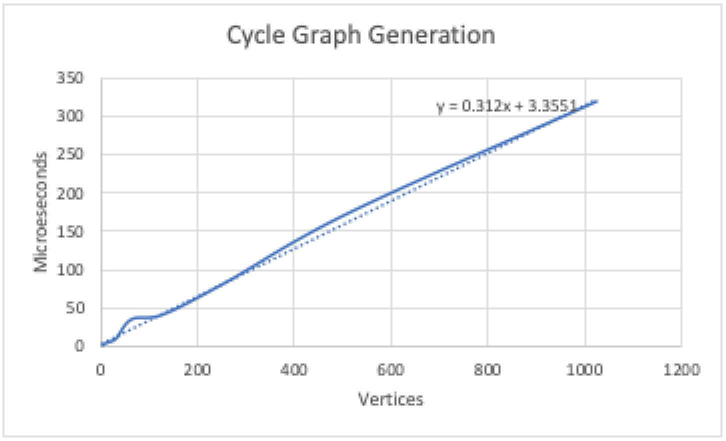


The graph and table give us a very good idea that this is a polynomial time implementation. The code itself must iterate through all V nodes in the graph V times. These two both direct to a polynomial time implementation, particularly a **Θ(V^2)** implementation. Comparing the times we can see the difference between two different subsequent rows that they are multiplied by around a factor of 4. It is not perfectly a factor of four due to other things happening on the machine, but it gets pretty close (717 * 4 = 2868 which is around only 300 microseconds off, and 38 microseconds * 4 is 152 microseconds which is only 3 microseconds off). Therefore, we can conclude that this implementation is **Θ(V^2).**

Cycle Graph Generation

Cycle Graph

V	Time (microseconds)
4	1
8	3
16	5
32	10
64	35
128	41
256	82
512	173
1024	318



Looking at the graph for cycle generation we can see a much more linear pattern than the complete graph! Comparing 16 to 32 nodes we can see very clearly that it takes exactly twice the amount of time to complete, and we see a similar trend for other vertices (exactly x2 between two subsequent vertices in many instances). This definitely looks like **Θ(V)** upon first glance. Looking at the code it only runs through the amount of vertices once and adds it to the graph. There is a constant set amount of operations performed for each vertex. Since we are doing a constant operation N times we can conclude that the cycle implementation is **Θ(V).**

Random Graph - Uniform

Time

Random - Uniform dist e = 0

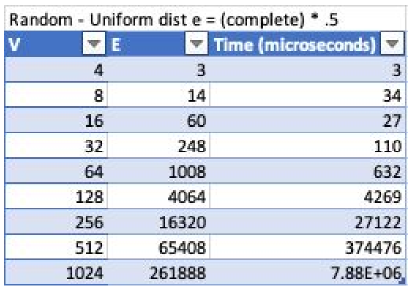
V	E	Time (microseconds)
4	0	4
8	0	1
16	0	0
32	0	1
64	0	3
128	0	3
256	0	5
512	0	14
1024	0	17

Random - Uniform dist e = V

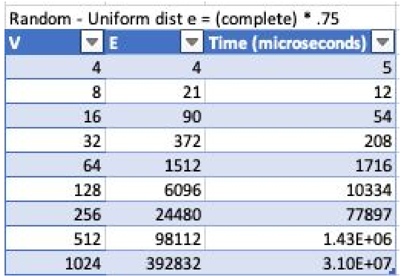
V	E	Time (microseconds)
4	4	5
8	8	9
16	16	8
32	32	15
64	64	55
128	128	61
256	256	137
512	512	233
1024	1024	462

Random - Uniform dist e = (complete) * .25

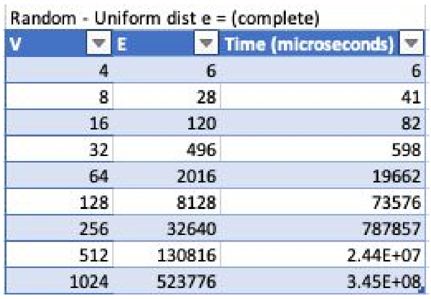
V	E	Time (microseconds)
4	1	2
8	7	24
16	30	14
32	124	53
64	504	492
128	2032	993
256	8160	6251
512	32704	57898
1024	130944	1.08E+06


Random - Uniform dist e = (complete) * .5

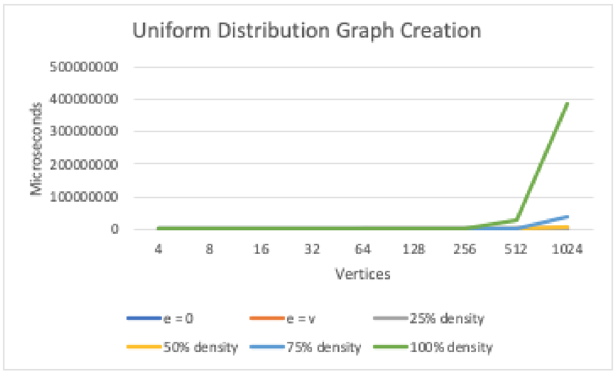
V	E	Time (microseconds)
4	3	3
8	14	34
16	60	27
32	248	110
64	1008	632
128	4064	4269
256	16320	27122
512	65408	374476
1024	261888	7.88E+06

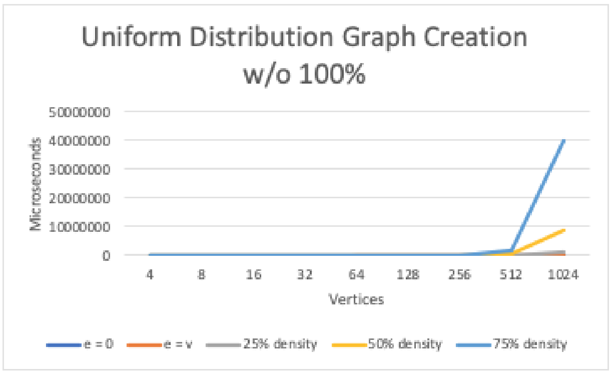

Random - Uniform dist e = (complete) * .75

V	E	Time (microseconds)
4	4	5
8	21	12
16	90	54
32	372	208
64	1512	1716
128	6096	10334
256	24480	77897
512	98112	1.43E+06
1024	392832	3.10E+07


Random - Uniform dist e = (complete)

V	E	Time (microseconds)
4	6	6
8	28	41
16	120	82
32	496	598
64	2016	19662
128	8128	73576
256	32640	787857
512	130816	2.44E+07
1024	523776	3.45E+08





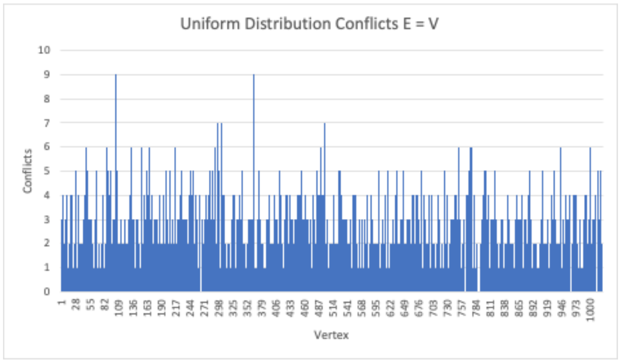
I found the timing for the random graphs very interesting. The comparisons between E = V and E at 100% density certainly shows some stark differences in time. I will now go into detail about the code and the times and what I believe the worst case, best case, and average case is.

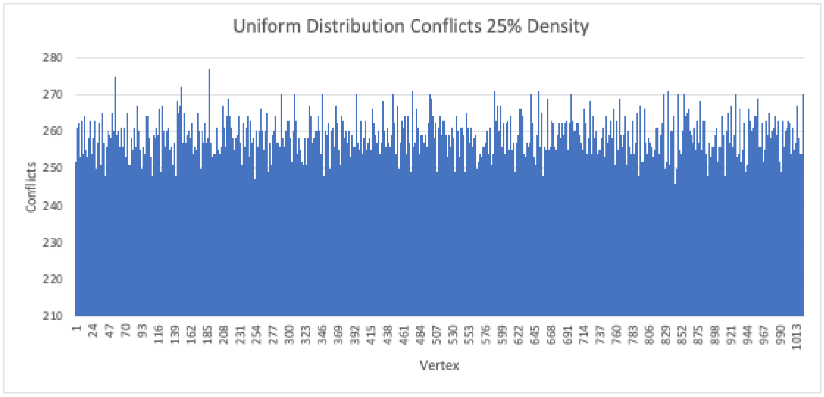
I believe that the best case implementation is **Ω(E).** This becomes clear after analyzing the E = V density. It very clearly goes up at a linear rate. For example, 128 vs 256 edges clearly only doubles in time which is directly related to doubling of edges. We see the same trend with 512 vs. 1024 edges. Also, there is no noticeable change anytime when e = 0. Therefore, we can assume that the best case implementation is **Ω(E)** because of this trend.

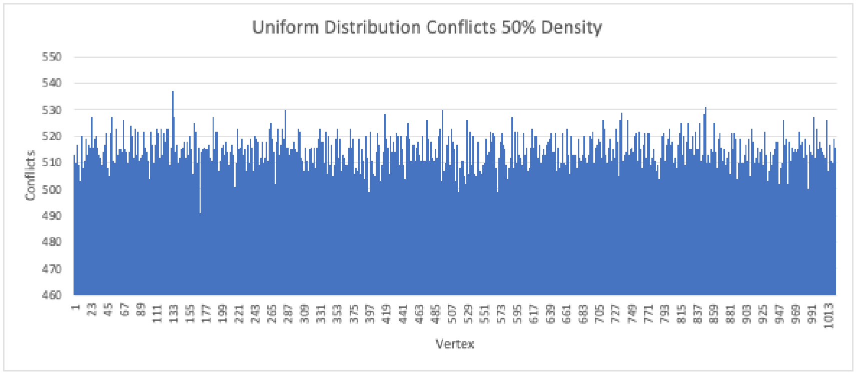
I believe that the worst case implementation is unfortunately **O(E!)**. This is mainly determined by looking at how the code works for generating complete graphs. The random generator has to end up going through all combinations randomly until eventually the right combination is picked. It might have to go through every possible combination of two vertices before it picks the right edge. This is *very* uncommon and is only really relevant for generating a complete graph, but using a random generator to generate a complete graph is inefficient because a complete graph generator would suit the same purpose in polynomial time.

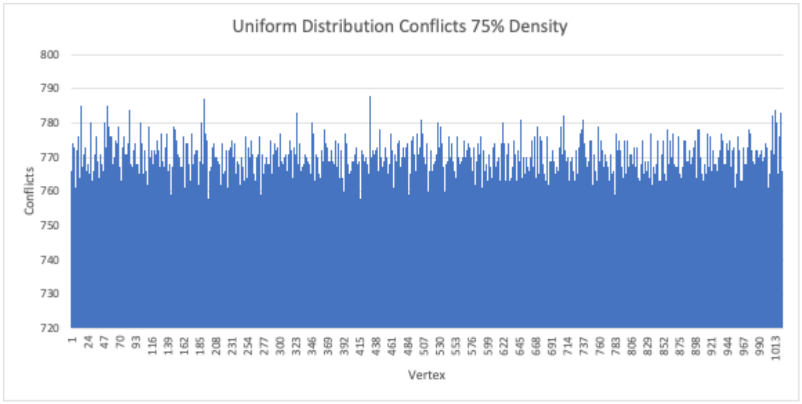
However, where things get interesting is when we talk about the average case. I believe the average case here is depending on how complete one wants their graph. For example, for smaller graphs for 32 vertices we see that the trend is acting somewhat polynomial. We see this by comparing 124 edges vs. 248 vs. 372 edges for different densities. The times for these edges are 53 microseconds, 110 microseconds, and 208 microseconds respectively. This would look linear on first glance since 124 * 2 is 248 vertices which is 53 * 2 = 106 microseconds which is only a slight 4 microseconds off from being linear, but 124 * 3 = 372 microseconds, but 53 * 3 = 159 microseconds, which is off by a nearly whole multiple of 2 to get to 208 microseconds. I had to dive into the code to get a better sense at what is going on. Since every new edge checks one vertex to see if it has been attached already, that adds more complexity for larger densities because each vertex will have more edges attached to it. We see similar cases for 504 edges vs. 1008 edges vs. 1512 edges with times 492 microseconds, 632 microseconds, and 1716 microseconds respectively. I am making the assumption that it is unlikely to generate a random *complete* graph because that raises the approximate time exceptionally sometime between 75% to 100% density. However, from densities 25% to 75% we can assume that the average case implementation is **Θ(E^2).**

Conflicts


Uniform Distribution Conflicts E = V
Conflicts
Vertex


Uniform Distribution Conflicts 25% Density
Conflicts
Vertex

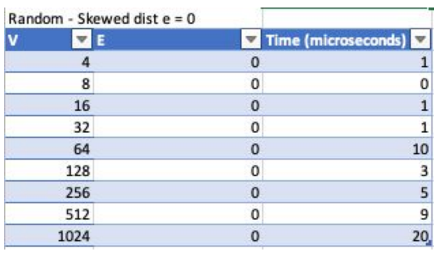

Uniform Distribution Conflicts 50% Density
Conflicts
Vertex


Uniform Distribution Conflicts 75% Density
Conflicts
Vertex

Conflicts look pretty representative of a uniform random distribution. I do not see any particular part of any density that sticks out as non-uniform. It looks similar to the uniform distribution in Generating Conflict Graphs. Therefore, we can assume that our uniform distribution works as expected.

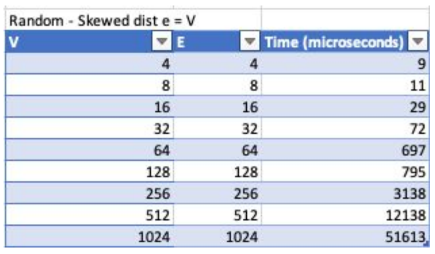
Random Graph - Skewed

Time



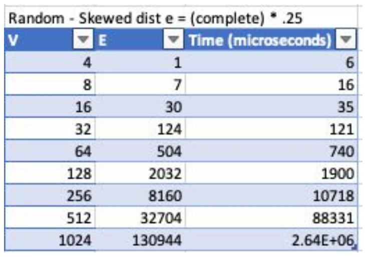
Random - Skewed dist e = 0

V	E	Time (microseconds)
4	0	1
8	0	0
16	0	1
32	0	1
64	0	10
128	0	3
256	0	5
512	0	9
1024	0	20



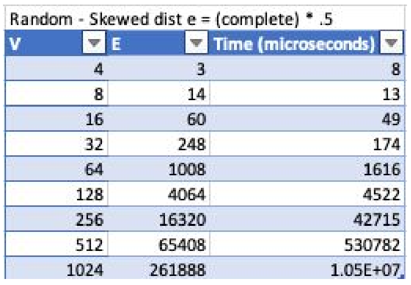
Random - Skewed dist e = V

V	E	Time (microseconds)
4	4	9
8	8	11
16	16	29
32	32	72
64	64	697
128	128	795
256	256	3138
512	512	12138
1024	1024	51613



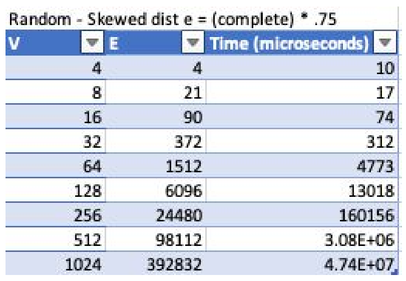
Random - Skewed dist e = (complete) * .25

V	E	Time (microseconds)
4	1	6
8	7	16
16	30	35
32	124	121
64	504	740
128	2032	1900
256	8160	10718
512	32704	88331
1024	130944	2.64E+06



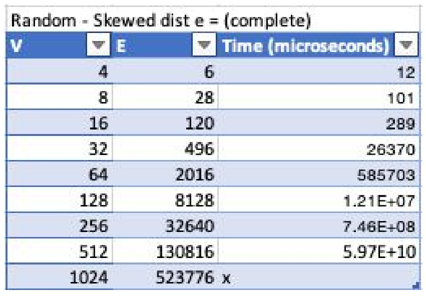
Random - Skewed dist e = (complete) * .5

V	E	Time (microseconds)
4	3	8
8	14	13
16	60	49
32	248	174
64	1008	1616
128	4064	4522
256	16320	42715
512	65408	530782
1024	261888	1.05E+07



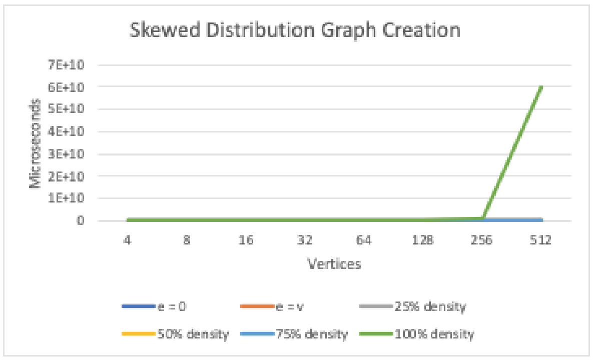
Random - Skewed dist e = (complete) * .75

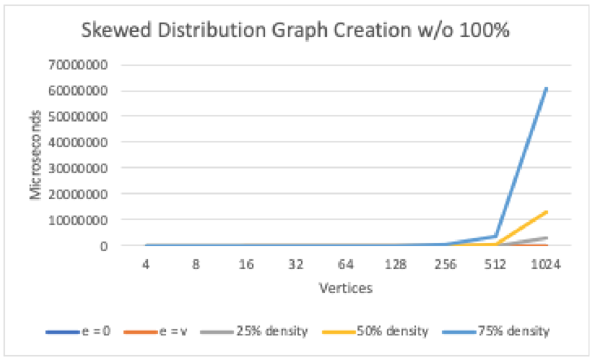
V	E	Time (microseconds)
4	4	10
8	21	17
16	90	74
32	372	312
64	1512	4773
128	6096	13018
256	24480	160156
512	98112	3.08E+06
1024	392832	4.74E+07



Random - Skewed dist e = (complete)

V	E	Time (microseconds)
4	6	12
8	28	101
16	120	289
32	496	26370
64	2016	585703
128	8128	1.21E+07
256	32640	7.46E+08
512	130816	5.97E+10
1024	523776	x





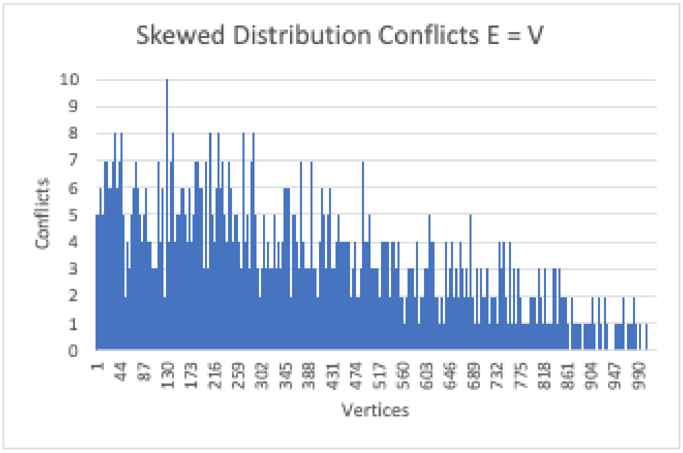
The times for the skewed graph are very similar to what we saw in Random Graph - Uniform. We will make very similar arguments in this section and I will back them up afterwards. This case is for a best case implementation of **Ω(E^2),** a worst case implementation of **O(E!)** and an average case implementation (between densities of 25% and 75%) of **Θ(E^2).**

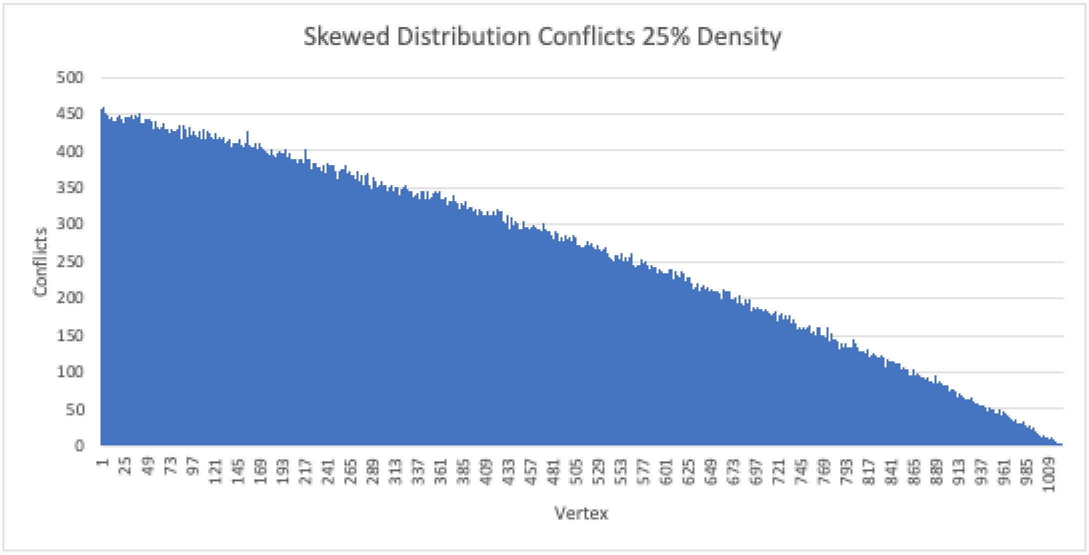
In terms of the best case one only needs to look at the E = 0 and the E = V denisites. We see an almost negligible amount of start time for the differences in the E = 0 table. For the E = V tables let's look at E = (256, 512, 1024) the times respectively in microseconds are (3138, 12138, 51613). Comparing 256 edges to 512 edges, if we are squared here then the time should be four times more. It is pretty close here. 3138 microseconds * 4 = 12552 microseconds and 12138 microseconds * 4 = 48552 microseconds for 512 vs. 1024 edges, and these are both pretty spot on for a best case implementation of **Ω(E^2).**

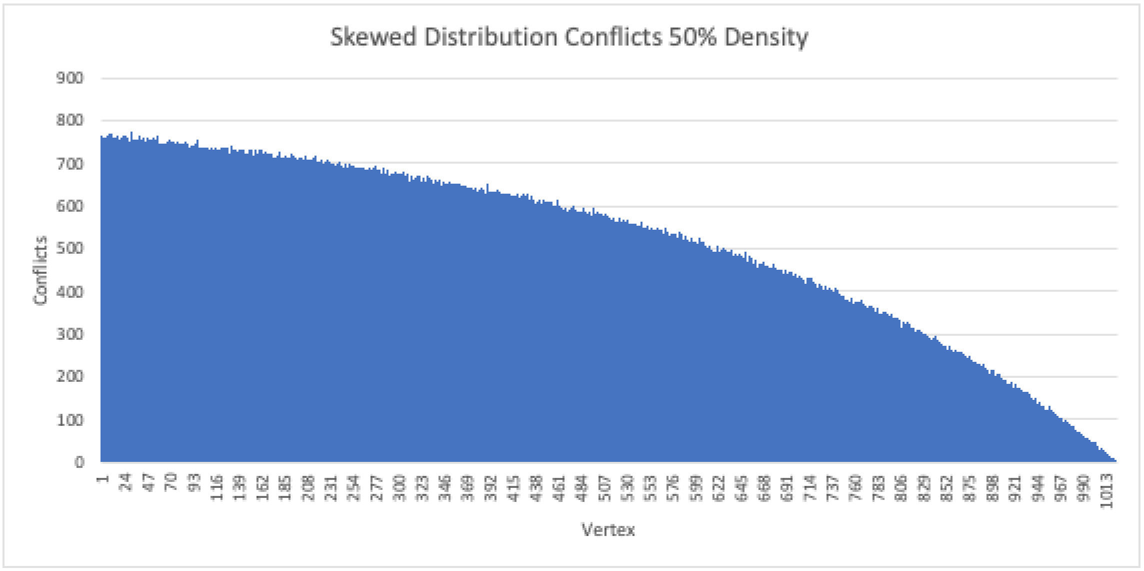
We can make the same argument here for the worst case as we did in the uniform section. **O(E!).** The implementation would have to iterate through every combination of pairs until eventually it finds the right one. This is common for finding the very last pair before making a complete graph! It is even worse on the skewed distribution because it is uncommon to get the high index vertices.

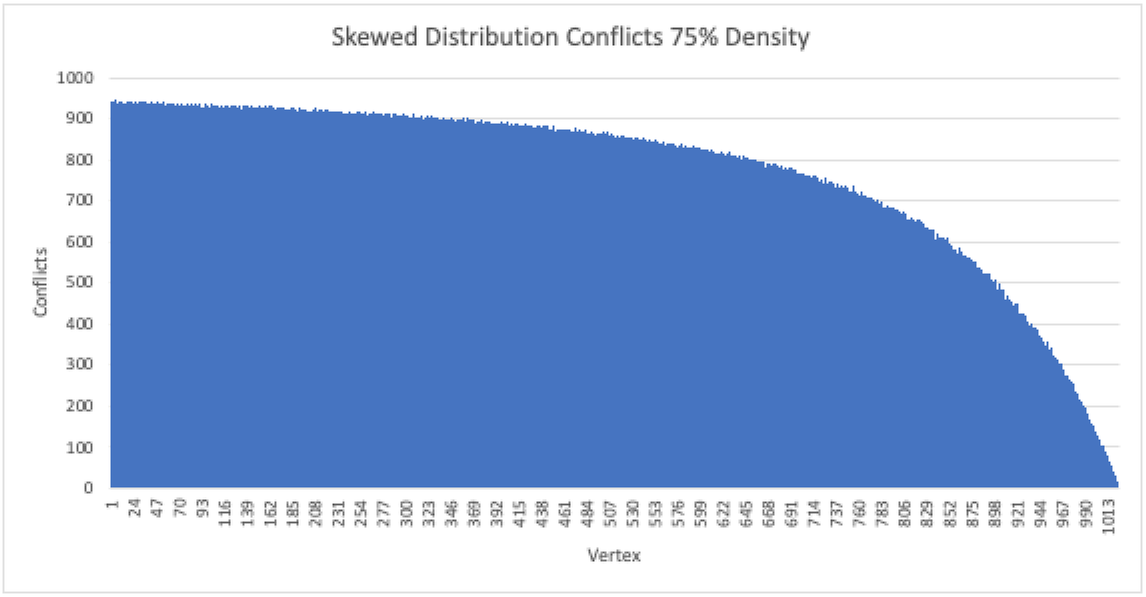
For the average case we will look at two different vertex counts(V = 64 and 128) at three different densities (25%, 50%, and 75%). For 64 vertices we see that it doubles at first and then increases at a polynomial rate (740, 1616, and 4773), but if it were O(E) it would have been (740, 1480, 2220). Therefore, we can conclude that it appears to be linear at low edges but it then grows at a polynomial rate when more edges are added to the graph. We see a similar trend for V = 128 (edges = {2032, 4064, 6096}, times = {1900, 4522, 13018} microseconds). It is slightly worse than linear at first between 2032 and 4064 but not quite enough to conclude that it is polynomial, but then we see polynomial growth between edges when comparing 2032 edges vs 6096 edges as it is growing at a rate higher than x3 (which would be a linear rate), so we can conclude we have polynomial growth. Therefore, just like the uniform distribution, we can assume that our average case for implementation is **Θ(E^2).**

Conflicts







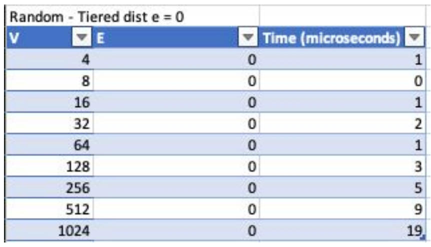


When E = V it does not paint a very clear picture about the distribution, but once we bump up the density it paints a very clear picture of the conflict distribution. At 25% it is almost identical to Generating Conflict Graphs, but the interesting thing is that the curve is more logarithmic as the density increases! I think this is because we simply just see the more likely vertices appear more often and it begins to contrast with the less likely vertices that almost never appear. I believe that once we max the amount of edges for many vertices we will begin to see the distribution curve back nicely, and eventually into a flat line once we reach a complete graph.

However, we can assume that the distribution is working as expected since it looks very similar to the example at 25% density.

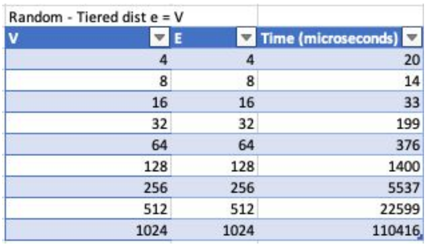
Random Graph - Tiered

Time



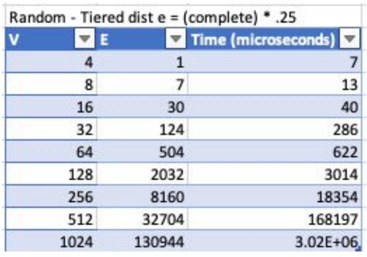
Random - Tiered dist e = 0

V	E	Time (microseconds)
4	0	1
8	0	0
16	0	1
32	0	2
64	0	1
128	0	3
256	0	5
512	0	9
1024	0	19



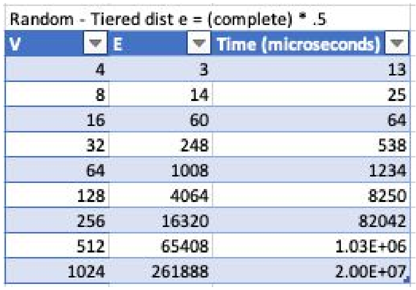
Random - Tiered dist e = V

V	E	Time (microseconds)
4	4	20
8	8	14
16	16	33
32	32	199
64	64	376
128	128	1400
256	256	5537
512	512	22599
1024	1024	110416



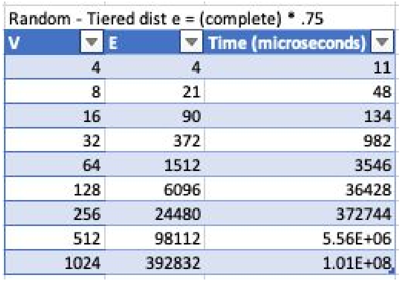
Random - Tiered dist e = (complete) * .25

V	E	Time (microseconds)
4	1	7
8	7	13
16	30	40
32	124	286
64	504	622
128	2032	3014
256	8160	18354
512	32704	168197
1024	130944	3.02E+06



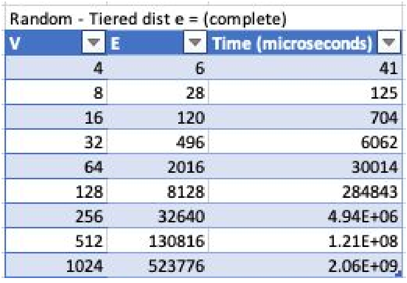
Random - Tiered dist e = (complete) * .5

V	E	Time (microseconds)
4	3	13
8	14	25
16	60	64
32	248	538
64	1008	1234
128	4064	8250
256	16320	82042
512	65408	1.03E+06
1024	261888	2.00E+07



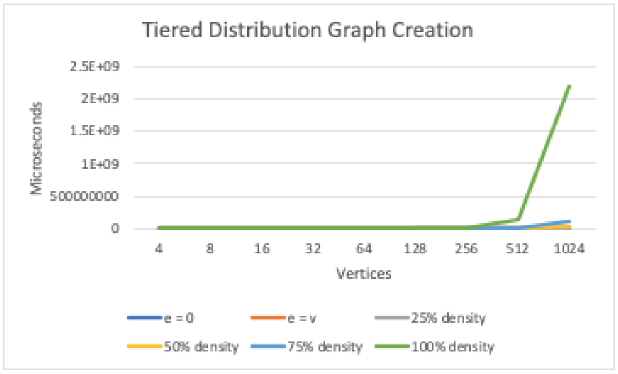
Random - Tiered dist e = (complete) * .75

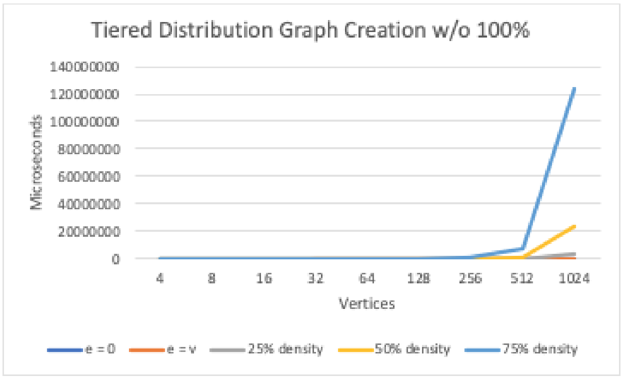
V	E	Time (microseconds)
4	4	11
8	21	48
16	90	134
32	372	982
64	1512	3546
128	6096	36428
256	24480	372744
512	98112	5.56E+06
1024	392832	1.01E+08



Random - Tiered dist e = (complete)

V	E	Time (microseconds)
4	6	41
8	28	125
16	120	704
32	496	6062
64	2016	30014
128	8128	284843
256	32640	4.94E+06
512	130816	1.21E+08
1024	523776	2.06E+09





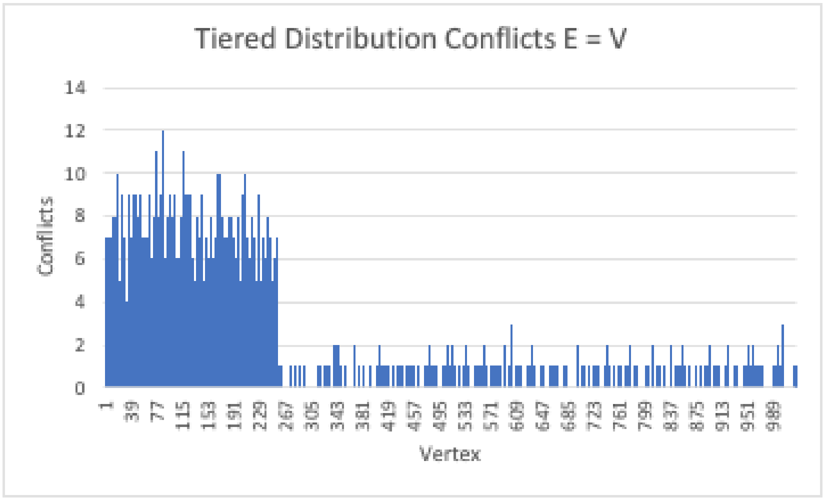
The tiered distribution is very similar to the skewed distribution. In fact I believe that all implementations share the same best, average, and worst case (**Ω(E^2), Θ(E^2), O(E!)** respectively**).**

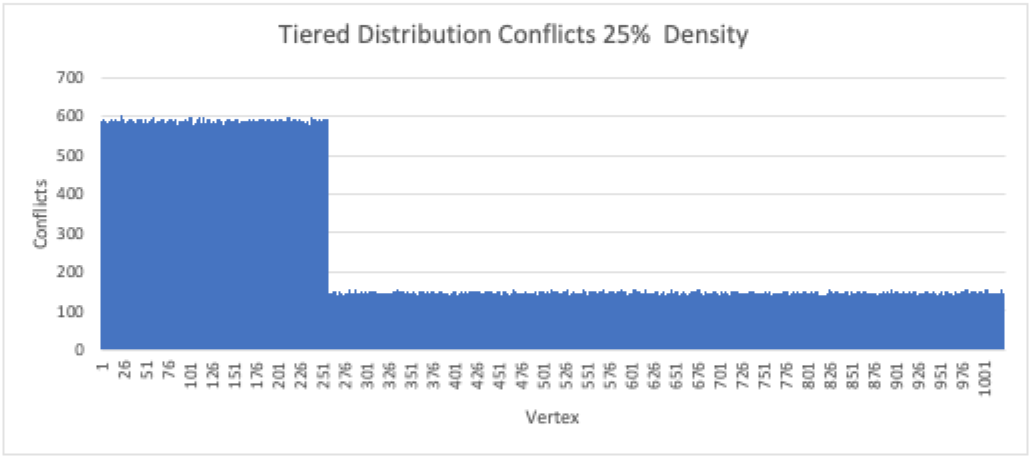
For our best case we can see yet again that for E = 0 there is basically negligible startup time. For E = V we see a similar trend to the skewed distribution. For E = 128 at time = 1400 microseconds if we multiply our time by 4 it equals 5600 microseconds. This is only 63 microseconds away from being exactly squared! Likewise we can see that 5537 microseconds * 4 = 22148 microseconds which is only about 400 microseconds away from being exactly square! This analysis paired with our knowledge of the code concludes that our best case for our implementation is **Ω(E^2).**

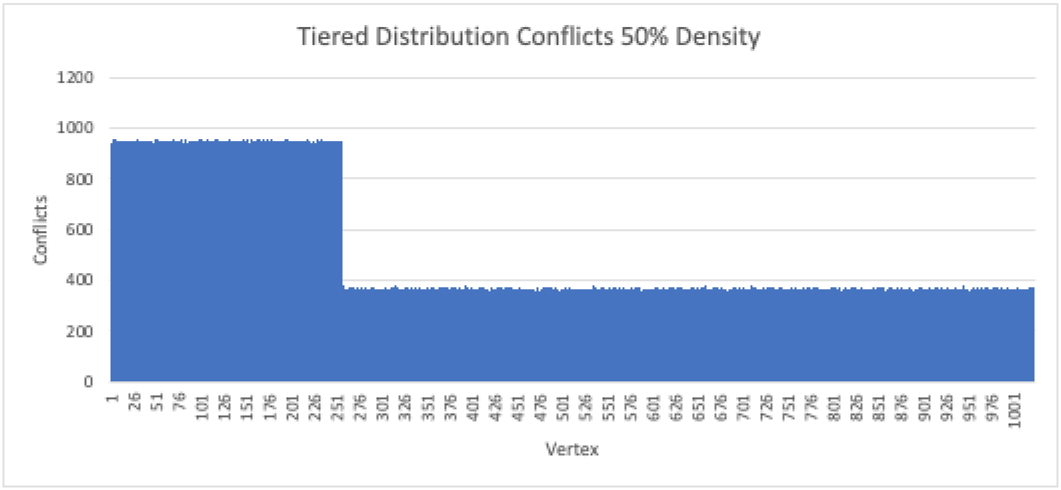
It is the same argument for the worst case that was made in the previous distributions, we will need to test every combination to get the some edges, it is likely this is the case for the last edge in a graph, and this can be amplified by the distribution. This is why we see our 100% density graph take significantly longer than the other densities, since it may need to throw out every edge combination before it arrives at the one correct combination to complete the graph.

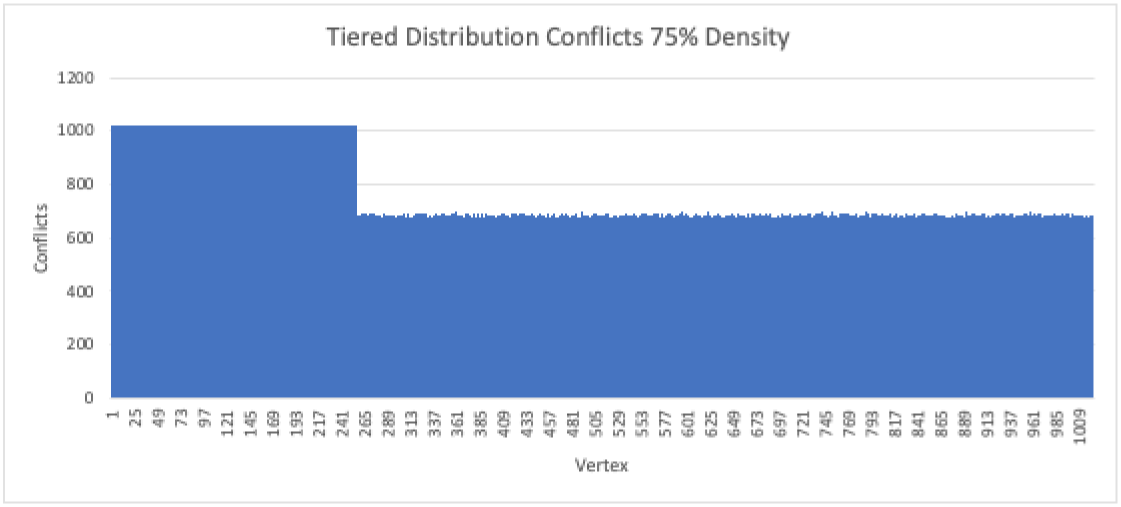
For the average case let's look at vertices 64 and 128. For vertex 64 (edges = {504, 1008, 1512} times (microseconds) = {622, 1234, 3546}) we see that as the amount of edges doubles, the time doubles as well, but once the edges triple we see the amount of time grow higher than a linear rate of x3, but it is less than squared (x16, 8064 microseconds). For 128 vertices (edges = {2032, 4064, 6096} times (microseconds) = {3014, 8250, 36428}) we can see that as the edges double, it is slightly larger than linear (x2 | 6028) but less than squared (x4 | 48224), and we see the same trend for 2032 edges vs 6096 edges. The time for these edges should be x3 different for linear and x16 different for squared complexity. However, 36428 microseconds (our result for E = 6096) fits between linear and squared. However, it is increasing polynomially! We can see this in our graphs, so we can assume that it must be an implementation of **Θ(E^2)** for average case.

Conflicts


Tiered Distribution Conflicts E = V
Conflicts
Vertex


Tiered Distribution Conflicts 25% Density
Conflicts
Vertex


Tiered Distribution Conflicts 50% Density
Conflicts
Vertex



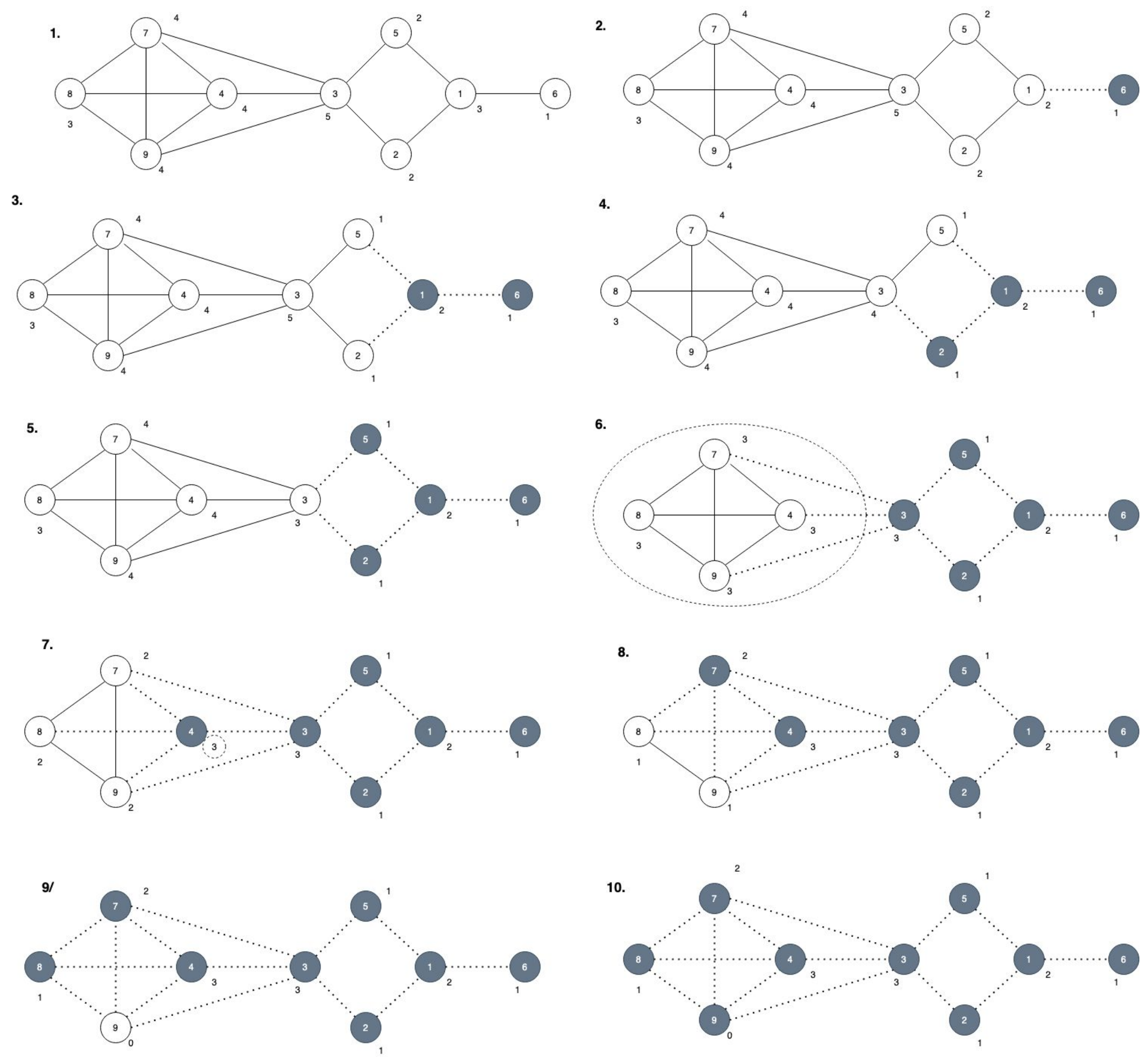
All densities thankfully look exactly like the sample graph! The one that interests me the most is the 75% density graph, particularly how the 20% portion of the tiered distribution makes up way more than 20% of the area. This is because the first quarter that receives 80% of the area has already maxed out their edges, so the only vertices that have edges left to add are the remaining three quarters of the graph. Overall, we can assume these distributions work as expected, but we should be careful using the tiered distribution for larger densities since it may not truly represent the 80%/20% difference in areas.

Vertex Ordering

In this section I will be going over the vertex ordering methods. For all methods I will be analyzing times for different kinds of graphs and deducing a timing analysis for the ordering methods. For smallest last vertex Ordering I will be going over additional information. All random graphs in this section are also generated using a uniform distribution. I will go through a step-by-step of a tangible example to show that it performs as expected. **Just like the previous section these numbers are averaged over five iterations. We are also not looking at random graphs at 100% density anymore since those are the same as complete graphs.**

Smallest Last Vertex

The following diagram is a numbered step-by-step ordering of smallest last vertex. A vertex turns grey when it is deleted from the graph.The number inside the vertex is the id. I assigned ids randomly to ensure it does not simply follow first fit to get the best ordering. The number next to the vertex is the current degree. Once a vertex is deleted its degree does not change. I circled both the terminal clique and the largest degree when deleted. We will go into detail in Smallest Last Vertex - Revisited about the significance of these numbers.



(A walkthrough of smallest last vertex Ordering)

Step 6 identifies the terminal clique. I have circled it to show its detection in the implementation. I have also circled the last occurrence of the highest degree when deleted. This is identified in step 7.

	Complete	
V	E	Time (microseconds)
4	6	1.8
8	28	6.6
16	120	18.2
32	496	59
64	2016	241
128	8128	961.8
256	32640	3882.6
512	130816	16025.8

	Cycle	
V	E	Time (microseconds)
4	4	1.4
8	8	4
16	16	5
32	32	8.6
64	64	17
128	128	32.8
256	256	58.8
512	512	141.6

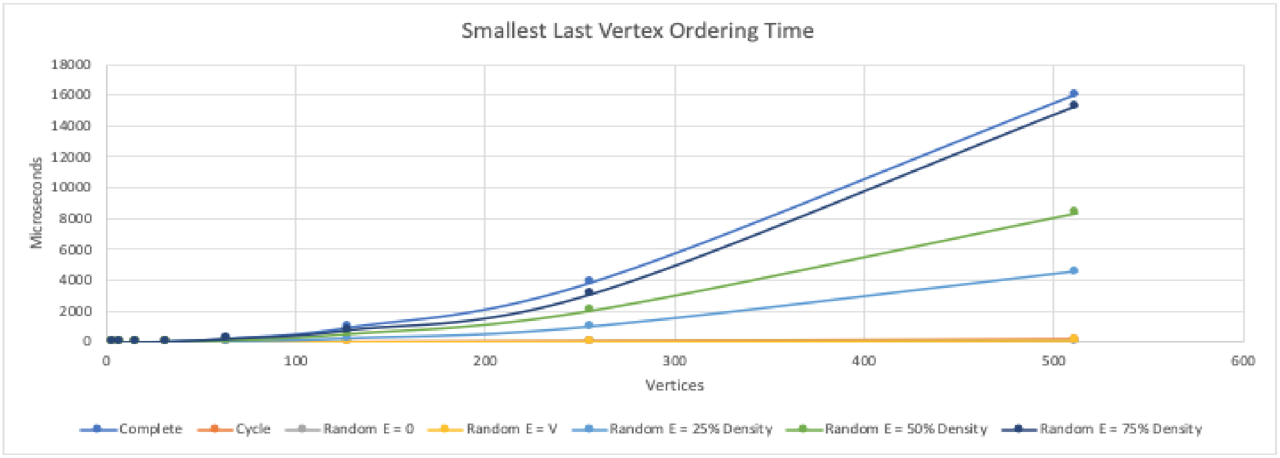
	Random E = 0	
V	E	Time (microseconds)
4	0	0.8
8	0	1.2
16	0	2.6
32	0	4
64	0	8
128	0	14
256	0	29.8
512	0	61.2

	Random E = V	
V	E	Time (microseconds)
4	4	2
8	8	2.8
16	16	4.8
32	32	10.4
64	64	16.4
128	128	31.4
256	256	63.2
512	512	124.6

	Random E = 25% Density	
V	E	Time (microseconds)
4	2	1
8	7	2.8
16	30	7.8
32	124	20.4
64	504	70.8
128	2032	266.8
256	8160	1031
512	32704	4578

	Random E = 50% Density	
V	E	Time (microseconds)
4	3	2.2
8	14	4.2
16	60	11.8
32	248	35.8
64	1008	130.8
128	4064	547.2
256	16320	2067.4
512	65408	8395.8

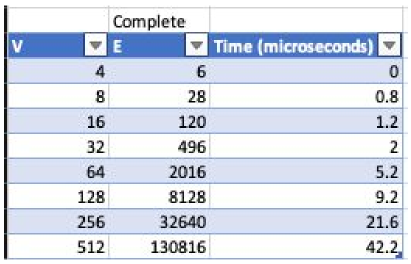
	Random E = 75% Density	
V	E	Time (microseconds)
4	5	2
8	21	4.6
16	90	14.2
32	372	50.2
64	1512	224.6
128	6096	792.6
256	24480	3125.4
512	98112	15274.2



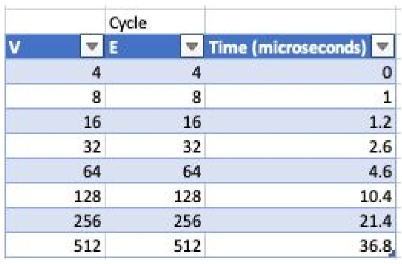
This smallest last vertex ordering method implementation runs in **Θ(E + V).** This algorithm follows the steps posted on the September 30th lecture. The steps that make it V + E are the steps that decrement the current degree of each adjacent non-deleted vertex when a vertex is deleted, and adjusts our list of edges to reflect that [1]. Looking at it analytically, it is a little easier to tell that it is **Θ(E + V).** Particularly if we look at the E = V table. Let's look at E = {64, 128, 256} and microseconds = {16.4, 31.5, 63.2}. At V = E our total amount of (V+E) calculations double for each new entry in the table, and it is clear that 16.4 microseconds is nearly half of 31.5 microseconds which is nearly half of 63.2 microseconds This is indicative of an edge related implementation. Therefore, analytically and through inspection of the code we can come to the conclusion that this implementation runs as **Θ(E + V).**

First Fit

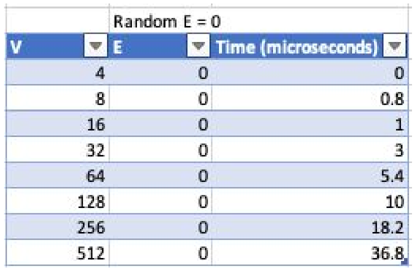
First fit is one of the most naive implementations for graph ordering. It simply orders the vertices in the order of their index, so in our graph it would just order them 1,2,3,4,...,n. I approximate this implementation runs in **Θ(V)** because it does not need to look at any edges: it just orders all vertices sequentially. Now let's look at the data.



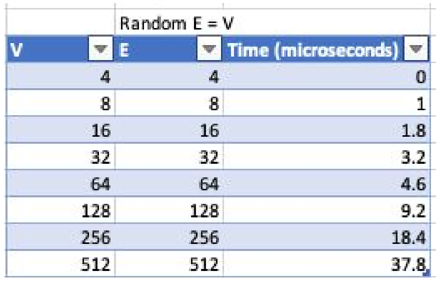
	Complete	
V	E	Time (microseconds)
4	6	0
8	28	0.8
16	120	1.2
32	496	2
64	2016	5.2
128	8128	9.2
256	32640	21.6
512	130816	42.2



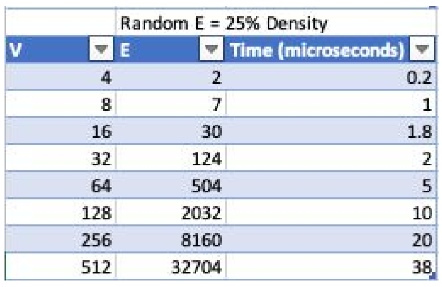
	Cycle	
V	E	Time (microseconds)
4	4	0
8	8	1
16	16	1.2
32	32	2.6
64	64	4.6
128	128	10.4
256	256	21.4
512	512	36.8



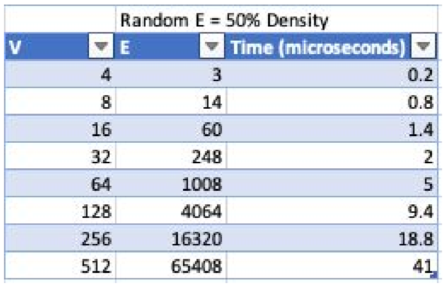
	Random E = 0	
V	E	Time (microseconds)
4	0	0
8	0	0.8
16	0	1
32	0	3
64	0	5.4
128	0	10
256	0	18.2
512	0	36.8



	Random E = V	
V	E	Time (microseconds)
4	4	0
8	8	1
16	16	1.8
32	32	3.2
64	64	4.6
128	128	9.2
256	256	18.4
512	512	37.8

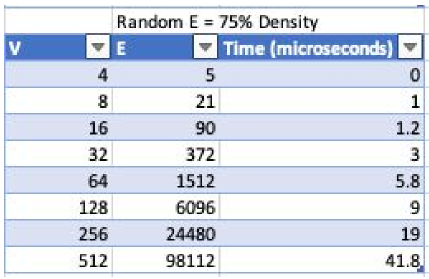


	Random E = 25% Density	
V	E	Time (microseconds)
4	2	0.2
8	7	1
16	30	1.8
32	124	2
64	504	5
128	2032	10
256	8160	20
512	32704	38



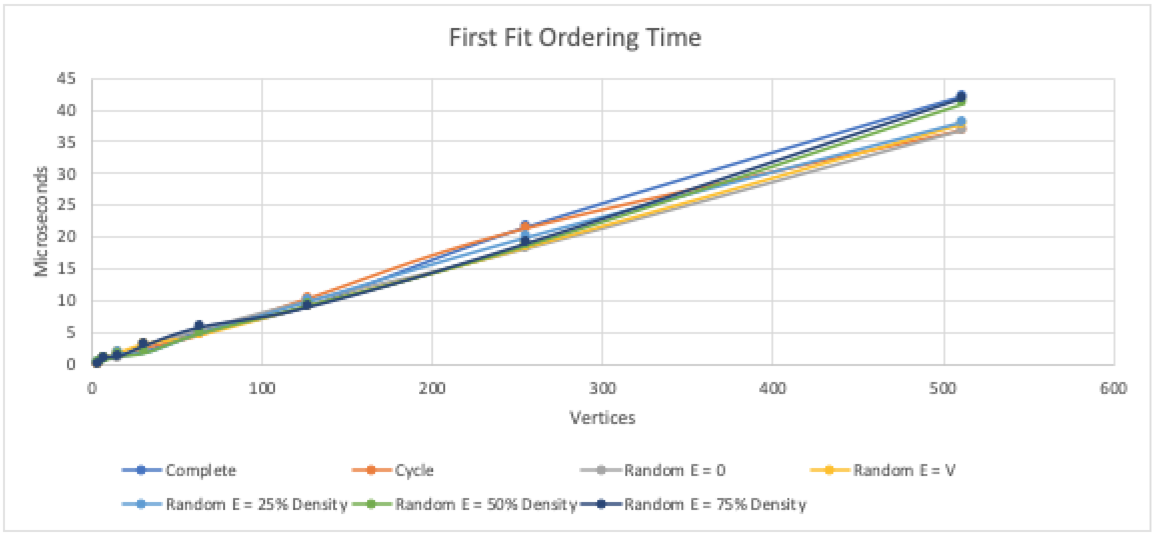
Random E = 50% Density

V	E	Time (microseconds)
4	3	0.2
8	14	0.8
16	60	1.4
32	248	2
64	1008	5
128	4064	9.4
256	16320	18.8
512	65408	41



Random E = 75% Density

V	E	Time (microseconds)
4	5	0
8	21	1
16	90	1.2
32	372	3
64	1512	5.8
128	6096	9
256	24480	19
512	98112	41.8



The tables and graphs at all densities are indicative of a linear constant. Since the tables are all around the same time regardless of the amount of edges we can conclude that vertices are the only significant variable. Looking at 25% density V = {64, 128, 256} and microseconds = {5, 10, 20} it really shows that it is following a linear trend. 5 microseconds * 2 is 10 microseconds and 10 microseconds * 2 = 20 microseconds. Therefore, we can conclude that my implementation of first fit runs in **Θ(V).**

Random

Random is slightly different from first fit. Instead of picking vertices sequentially, it randomises the vertices and then picks the shuffled vertices sequentially. Random ordering has no concept of edges, and my implementation shuffles all vertices and then picks them. Therefore, I approximate my implementation of a random ordering runs in **Θ(V).** Now let's look at the data.

	Complete	
V	E	Time (microseconds)
4	6	1
8	28	1.6
16	120	2
32	496	4.2
64	2016	7.4
128	8128	20.2
256	32640	25.2
512	130816	50.2

	Cycle	
V	E	Time (microseconds)
4	4	1
8	8	2
16	16	2
32	32	6.4
64	64	6.2
128	128	12
256	256	23.6
512	512	67.2

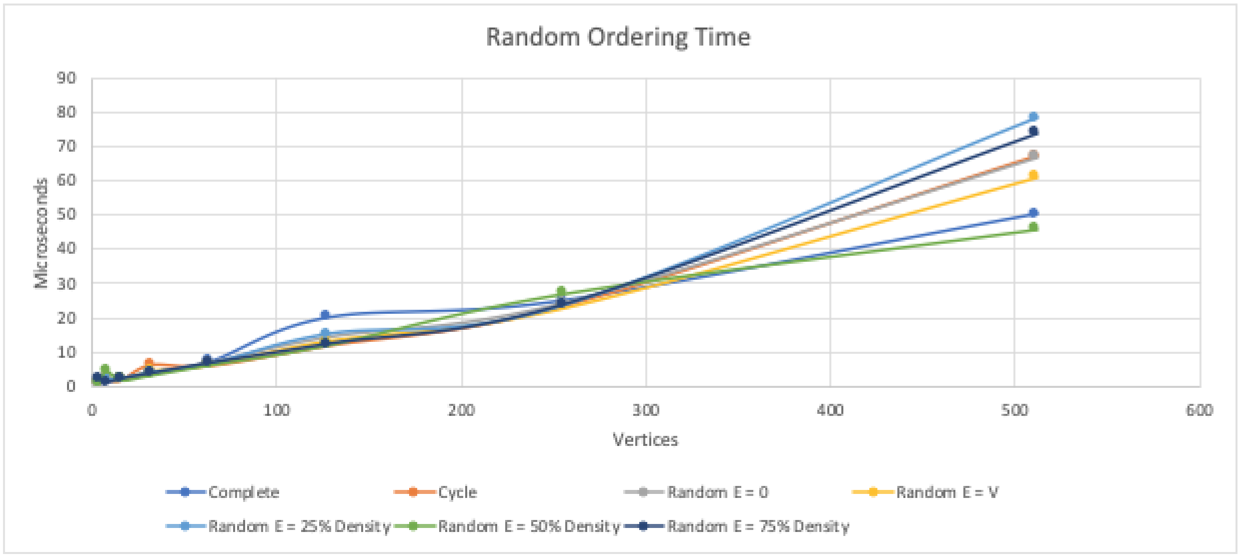
	Random E = 0	
V	E	Time (microseconds)
4	0	1
8	0	2
16	0	2
32	0	4
64	0	7.2
128	0	14.6
256	0	24.6
512	0	66.8

	Random E = V	
V	E	Time (microseconds)
4	4	1
8	8	2.2
16	16	2
32	32	4.6
64	64	6.4
128	128	13.4
256	256	23
512	512	61

	Random E = 25% Density	
V	E	Time (microseconds)
4	2	1.2
8	7	2
16	30	2.2
32	124	4
64	504	6.4
128	2032	15.2
256	8160	23.6
512	32704	78.2

	Random E = 50% Density	
V	E	Time (microseconds)
4	3	1
8	14	4.6
16	60	2
32	248	3.4
64	1008	6.4
128	4064	12.2
256	16320	27.2
512	65408	46

	Random E = 75% Density	
V	E	Time (microseconds)
4	4.5	2.2
8	21	1.2
16	90	2.2
32	372	3.8
64	1512	6.8
128	6096	12.4
256	24480	24
512	98112	74



We see that random has every edge density packed together very tightly. This is very similar to first fit. We also have to keep in mind how low 70 or 80 microseconds really is. This is very few microseconds compared to other orderings like smallest last vertex ordering. I assume the blips in the complete graph are due to something on my machine happening in the background. Looking Random E = V with vertices = {64, 128, 256} and microseconds = {6.4, 13.4, 23} we see another linear comparison since 6.4 microseconds * 2 is very close to 13.4 microseconds just as 13.4 microseconds * 2 is close to 23 microseconds. All graphs exhibit this same linear comparison with some blips in times that won't make it perfect. Something to note is that there is not one clear winner in the pack because it keeps switching around in the graph, and this shows that edges are not relevant in the generation of this graph. Because of the strong linearity in the code and in the data we will draw the conclusion that the random ordering implementation runs in **Θ(V).**

Smallest Original Degree Last

Smallest original degree last is very similar to smallest last vertex, but it does not look at edges for remaining vertices. Instead it goes solely off of the degree that each vertex has upon graph creation. Therefore, there is no need to keep up with degrees and lists of edges unlike smallest last vertex. However, since it does need to know the current degree for each vertex I approximate my implementation can not be any faster than **Θ(E + V).** Now let's look at the data.

Complete

V	E	Time (microseconds)
4	6	0.8
8	28	0.6
16	120	1.2
32	496	3.2
64	2016	6.6
128	8128	21.2
256	32640	43.6
512	130816	96

Cycle

V	E	Time (microseconds)
4	4	0.8
8	8	0.8
16	16	1.8
32	32	2.8
64	64	5
128	128	10.4
256	256	22.4
512	512	47.8

Random E = 0

V	E	Time (microseconds)
4	0	0.2
8	0	0.2
16	0	1.4
32	0	3.4
64	0	7
128	0	11.8
256	0	21.6
512	0	44.8

Random E = V

V	E	Time (microseconds)
4	4	0
8	8	0.8
16	16	1
32	32	3
64	64	5.2
128	128	10.8
256	256	20.2
512	512	43.4

Random E = 25% Density

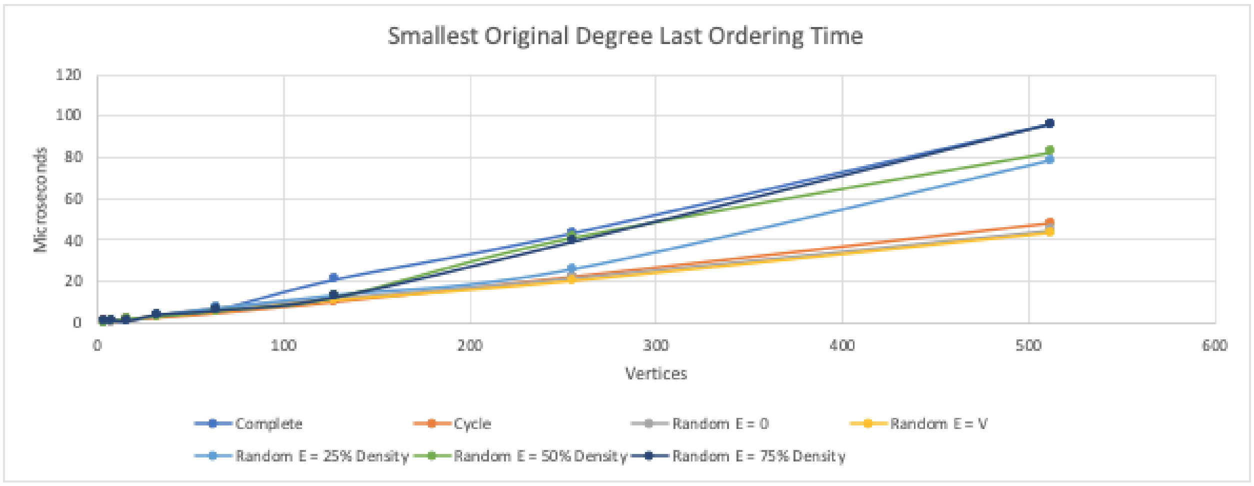
V	E	Time (microseconds)
4	2	0
8	7	1
16	30	1
32	124	3.2
64	504	6.8
128	2032	13.2
256	8160	25.8
512	32704	78.2

Random E = 50% Density

V	E	Time (microseconds)
4	3	0.2
8	14	1
16	60	1.8
32	248	3
64	1008	5.8
128	4064	12.4
256	16320	41.2
512	65408	82.4

Random E = 75% Density

V	E	Time (microseconds)
4	5	0.4
8	21	1.2
16	90	1
32	372	4
64	1512	6
128	6096	12.6
256	24480	39.4
512	98112	96



Unlike what we saw in Random and First Fit there is a clear distinction between the low edge graphs and the high edge graphs, and our complete graph is at the top of the list. Therefore, we can make the assumption that there is an edge relation in our time analysis. For analysis let's look at the E = V table at E = {64, 128, 256} and microseconds = {5.2, 10.8, 20.2}. At 64 vertices it is half the amount of edges + vertices than 128 which is half the edges + vertices of 256 (128, 256, 512) and we see that is the case for timing as well. This shows us that our time is edge related, but also non-polynomial. Therefore we can conclude that my smallest original degree last ordering implementation runs in **Θ(E + V).**

Incidence Degree

This ordering method orders vertices by the largest number of deleted vertices [2]. This vertex method tends to pick out large cliques first, and then focus on smaller and less connected vertices. It works very similarly to the smallest last vertex, but instead of using a second list for current degree we use a second list for "incidence," or how many adjacent deleted vertices are for each edge. We then traverse down the list rather than up. After inspection of code I approximate that this ordering implementation runs in **Θ(E + V).** Now let's look at the data.

	Complete	
V	E	Time (microseconds)
4	6	3
8	28	5.8
16	120	17.2
32	496	63
64	2016	242
128	8128	949.8
256	32640	4284
512	130816	16492

	Cycle	
V	E	Time (microseconds)
4	4	2.4
8	8	10.4
16	16	6.6
32	32	11.6
64	64	20.6
128	128	40.2
256	256	84
512	512	165

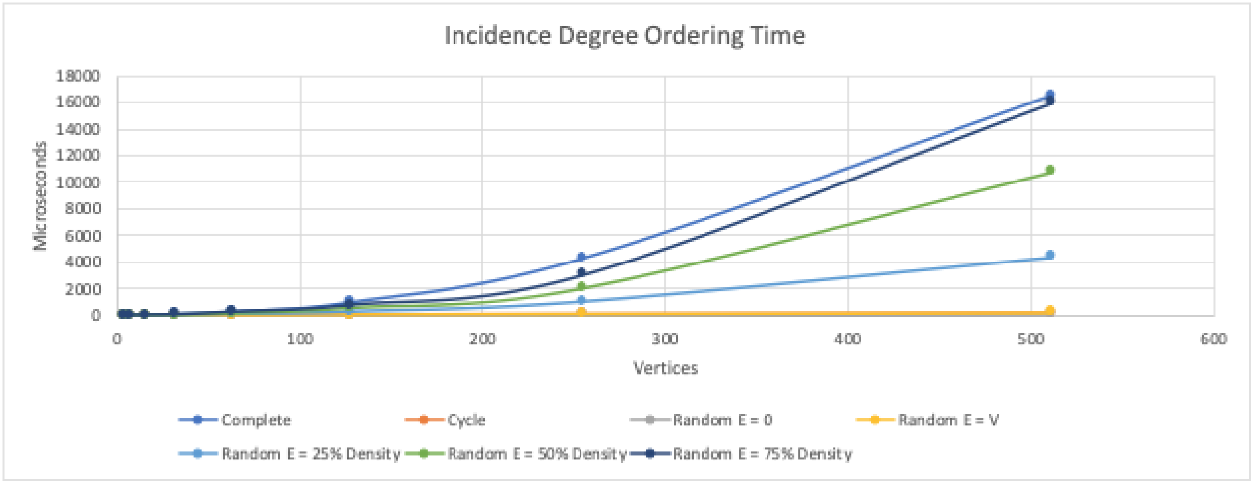
	Random E = 0	
V	E	Time (microseconds)
4	0	2
8	0	2.6
16	0	4
32	0	8.6
64	0	13.4
128	0	26
256	0	51.8
512	0	109.4

	Random E = V	
V	E	Time (microseconds)
4	4	2.6
8	8	4.4
16	16	7.8
32	32	11.6
64	64	22.4
128	128	42.8
256	256	96.2
512	512	227.2

	Random E = 25% Density	
V	E	Time (microseconds)
4	2	1.8
8	7	3
16	30	9.6
32	124	26.6
64	504	101
128	2032	269.8
256	8160	1040.6
512	32704	4419.2

	Random E = 50% Density	
V	E	Time (microseconds)
4	3	2.4
8	14	4.2
16	60	12.2
32	248	38
64	1008	137.2
128	4064	533.2
256	16320	2042.8
512	65408	10812

	Random E = 75% Density	
V	E	Time (microseconds)
4	5	3.2
8	21	6.6
16	90	18
32	372	60.4
64	1512	260.2
128	6096	762
256	24480	3032.4
512	98112	15945.4



We see a very similar trend to graph ordering methods that are E + V. To confirm That we are running in **Θ(E + V)** time let's compare the cycle and the complete graphs. For the complete graph comparing V, E = 256, 32640 vs. V, E = 512, 130816 the difference in V + E for these two pairs is approximately 4x, therefore the time should increase by 4. Thankfully this works! 4284 microseconds * 4 = 17136 microseconds which is about 500 microseconds off from the actual 512 vertex result of 16492 microseconds. Additionally, we see a similar analysis for cycle graphs. Comparing 256 vertices vs. 512 vertices should be an increase of x2 since we are doubling the combination of edges and vertices. 84 microseconds * 2 = 168 microseconds. This is only 3 microseconds off from our result at 512! Therefore, we can conclude that this implementation of incidence degree ordering is **Θ(E + V).**

BFS

BFS is the common searching algorithm that searches a graph through a traditionally wide implementation [3]. We used BFS here as another perspective to see if keeping nearby vertices in order would be important for coloring. However, we will discuss that more in the final section.

Complete

V	E	Time (microseconds)
4	6	2.4
8	28	3.4
16	120	6.2
32	496	11.6
64	2016	23
128	8128	48
256	32640	89
512	130816	182

Cycle

V	E	Time (microseconds)
4	4	2.4
8	8	3.8
16	16	7.2
32	32	13.6
64	64	24.8
128	128	43.4
256	256	87
512	512	199

Random E = 0

V	E	Time (microseconds)
4	0	1.2
8	0	2.4
16	0	4
32	0	7.8
64	0	13.8
128	0	24
256	0	47.8
512	0	96

Random E = V

V	E	Time (microseconds)
4	4	2.6
8	8	4.2
16	16	7.2
32	32	13.2
64	64	24.4
128	128	47.2
256	256	93.8
512	512	205.8

Random E = 25% Density

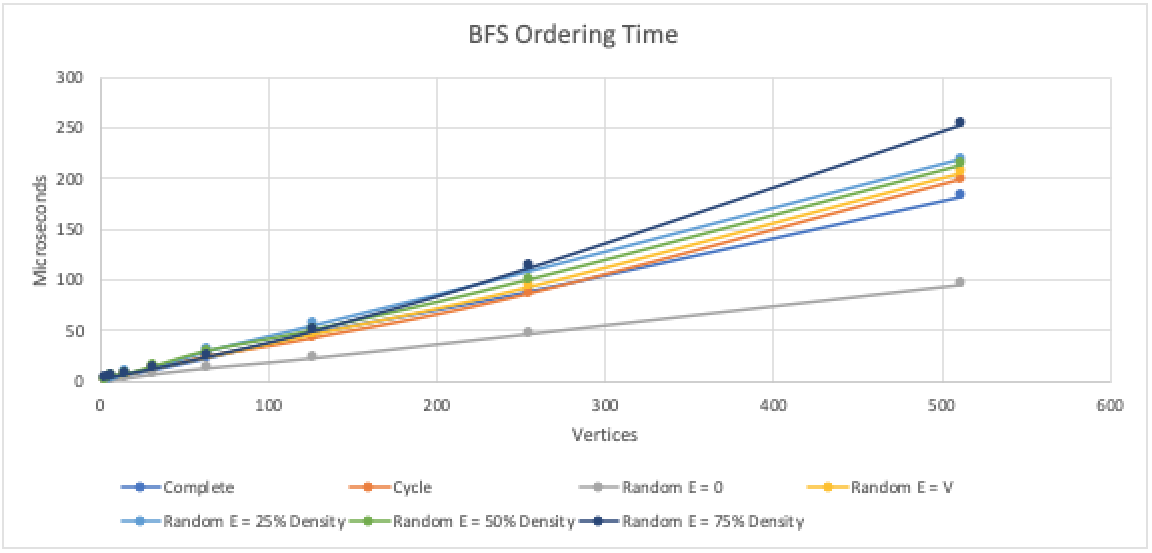
V	E	Time (microseconds)
4	2	1.2
8	7	4
16	30	8.6
32	124	14
64	504	30.6
128	2032	56.4
256	8160	109.4
512	32704	219

Random E = 50% Density

V	E	Time (microseconds)
4	3	2
8	14	5.6
16	60	7.4
32	248	15
64	1008	30.2
128	4064	51.4
256	16320	101
512	65408	213.6

Random E = 75% Density

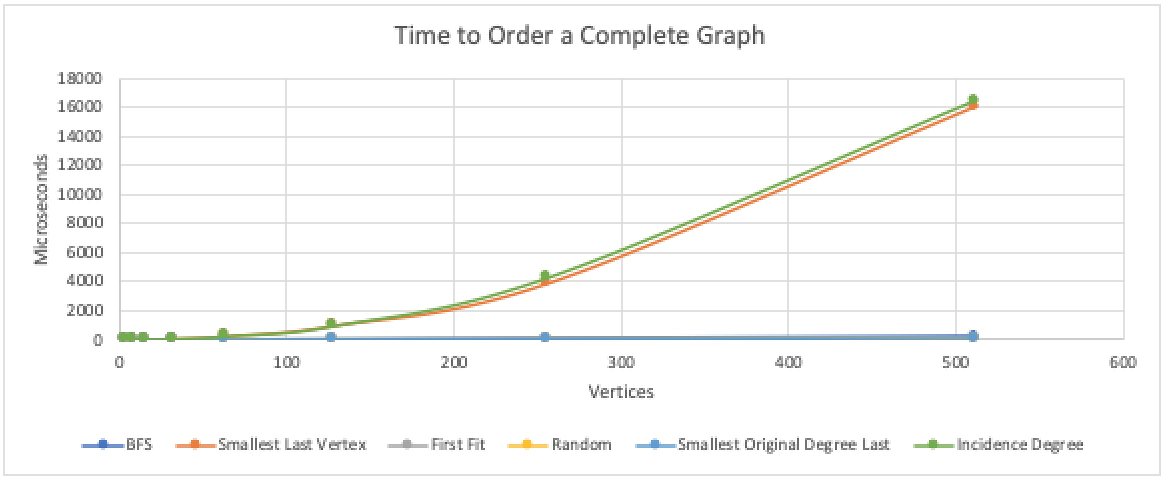
V	E	Time (microseconds)
4	5	2.8
8	21	5.2
16	90	7.8
32	372	13.2
64	1512	25
128	6096	50.8
256	24480	113.2
512	98112	253.8

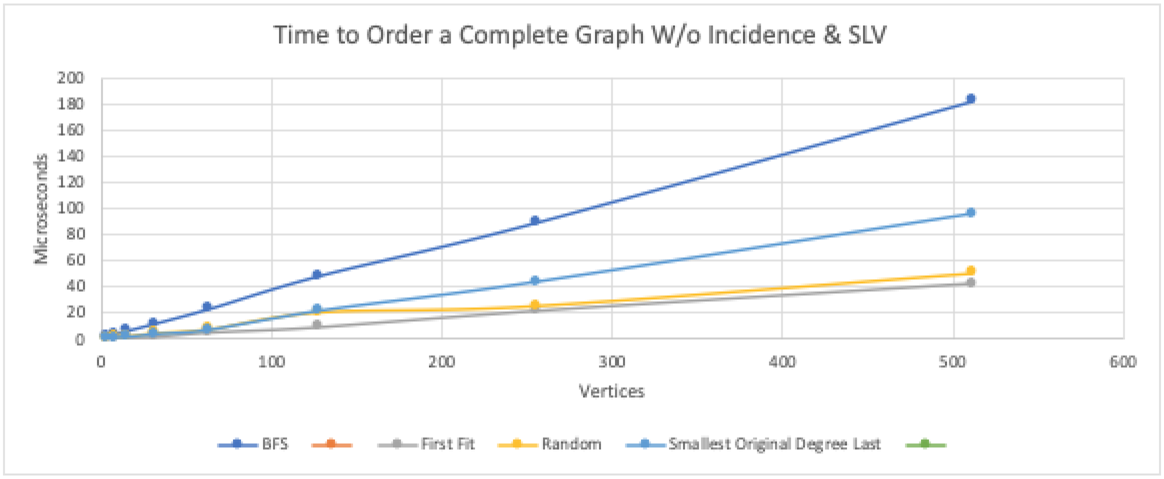


The graph here appears that there is no strong relationship to the number of edges. This is also by the fact that the 75% density graph took the longest to complete. This is not indicative of the number of edges. However, something interesting is how the empty graph was the quickest. I believe this is because BFS has different conditions for mutually exclusive sets, and this logic is faster than traditional BFS logic. Comparing the vertices in the complete graph, they mostly increase by a factor of 2 for each iteration. This shows that this is a vertex relationship because edges increase by a factor of ~4. Therefore, we can make the assumption that this BFS ordering implementation runs in **Θ(V).**

Overall

This section gives us a very good idea about the implementation complexity of our algorithms. Later on in Vertex Ordering Capabilities we will look at how well these algorithms do in terms of coloring. We can suspect that the more involved algorithms like incidence degree and smallest last vertex will give us the fewest amount of colors used, but we will discuss that later on.

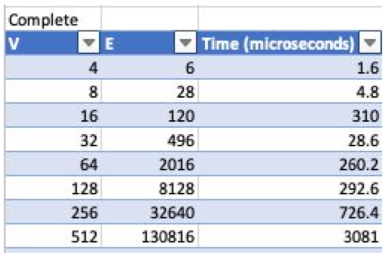




We can see that smallest last vertex and incidence degree are outliers when it comes to time to complete, while the other 4 methods take much less time. If a developer is in a pinch with time constraints for ordering methods perhaps it would be more appropriate to choose one of the four faster ordering implementation. Later on we will go in detail about which methods are the best at limiting the number of colors.

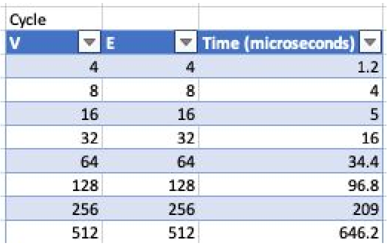
Coloring Algorithm

In this section we will be looking at the coloring algorithm. The coloring algorithm is a simple greedy algorithm. It assigns the first available color to each vertex. All color orderings were assigned with smallest last vertex ordering. All timings were averaged over 5 attempts.



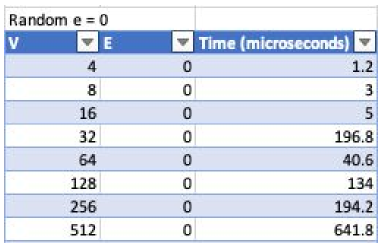
Complete

V	E	Time (microseconds)
4	6	1.6
8	28	4.8
16	120	310
32	496	28.6
64	2016	260.2
128	8128	292.6
256	32640	726.4
512	130816	3081



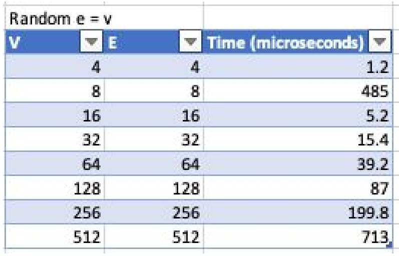
Cycle

V	E	Time (microseconds)
4	4	1.2
8	8	4
16	16	5
32	32	16
64	64	34.4
128	128	96.8
256	256	209
512	512	646.2



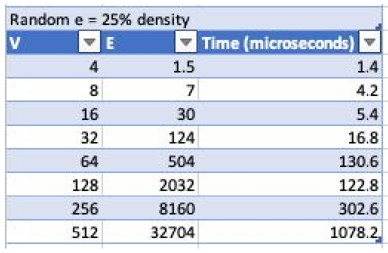
Random e = 0

V	E	Time (microseconds)
4	0	1.2
8	0	3
16	0	5
32	0	196.8
64	0	40.6
128	0	134
256	0	194.2
512	0	641.8



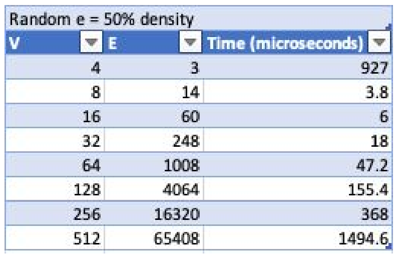
Random e = v

V	E	Time (microseconds)
4	4	1.2
8	8	485
16	16	5.2
32	32	15.4
64	64	39.2
128	128	87
256	256	199.8
512	512	713



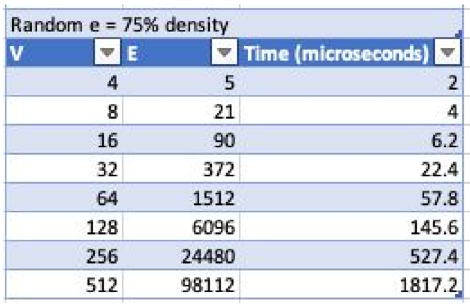
Random e = 25% density

V	E	Time (microseconds)
4	1.5	1.4
8	7	4.2
16	30	5.4
32	124	16.8
64	504	130.6
128	2032	122.8
256	8160	302.6
512	32704	1078.2



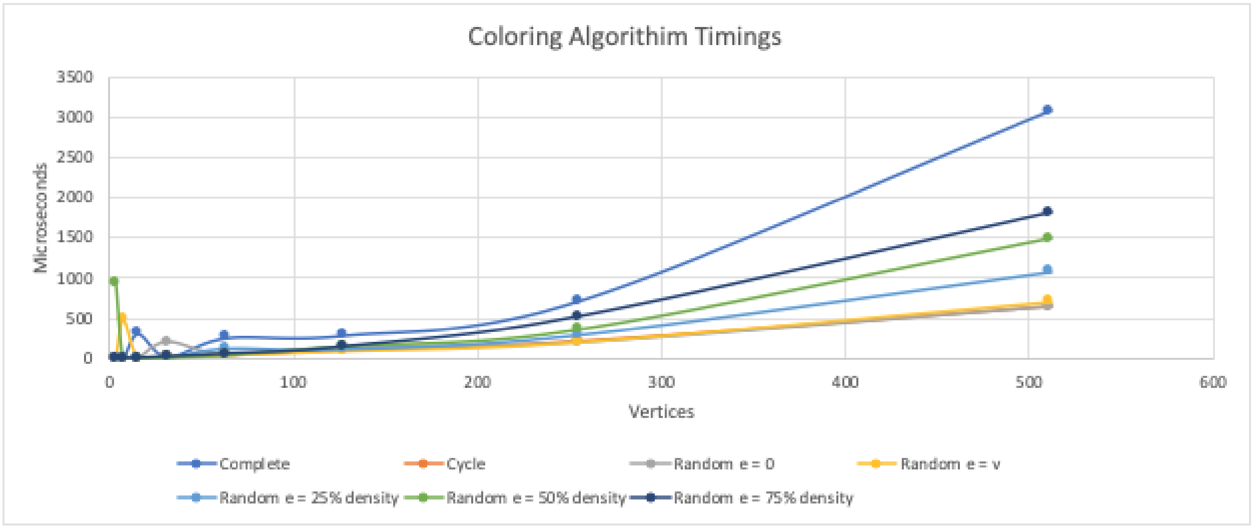
Random e = 50% density

V	E	Time (microseconds)
4	3	927
8	14	3.8
16	60	6
32	248	18
64	1008	47.2
128	4064	155.4
256	16320	368
512	65408	1494.6



Random e = 75% density

V	E	Time (microseconds)
4	5	2
8	21	4
16	90	6.2
32	372	22.4
64	1512	57.8
128	6096	145.6
256	24480	527.4
512	98112	1817.2



At first glance the graph looks like the coloring is directly related to the amount of edges on a graph. The graph with the most edges (complete) is followed by the second most dense graph (75% density) followed by 50% density. Therefore, we can assume there is some correlation between edges and time.

Upon inspection of the code and the times. We can make the assumption that the timing complexity for the coloring implementation is **Ω(E + V) and O(V^2 + E).** We have our best case implementation of **Ω(E + V)** for a few special case graphs. One example of this would be either an empty graph (one with no edges) or a graph that is very sparsely connected like when E = V. The inspection of the code shows that coloring a complete graph is **O(V^2 + E)** because each edge is connected to every vertex. This means we need to traverse every edge and visit every vertex by a squared polynomial while also traversing all edges.

Looking at the tables to ensure that we are between **Ω(E + V) and O(V^2 + E),** our random E = V graph (E = {64, 128, 256} microseconds = {39.2, 87, 199.8}). Comparing 64 edges to 128 edges the total increase in vertices and edges is x2 (64 + 64 vs. 128 + 128) so the time should be x2 as well. We can see that it is pretty close to x2 since 39.2 microseconds * 2 = 78.4 microseconds microseconds which is about 9 microseconds off from being **Ω(E + V)**. We can see the same pattern with 128 edges vs. 256 edges. The increase is x2 and 87 microseconds * 2 = 174 microseconds which is about 25 microseconds off from being **Ω(E + V)**. This is pretty close!

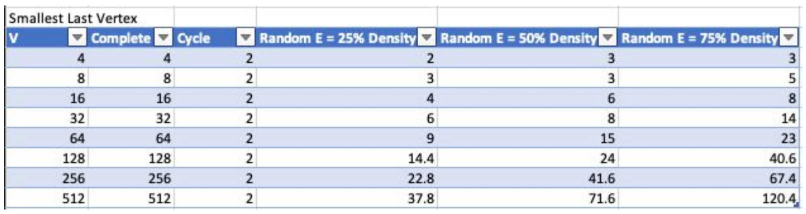
Now, analytically looking at **O(V^2 + E)** we will look solely at the complete graph. In particular we will be looking at V = {256, 512}, microseconds = {726.4, 3081}, and V^2 + E (computations) = {98176, 392960}. We can make the assumption here that 98176 computations multiplied by a factor of 4 would bring it very close to 392960 computations (To be completely accurate we would need to multiply 98176 computations by 4.0026). Therefore, the times should also increase by a factor of four. We can see that this is the case. Multiplying 726.4

microseconds by 4 gives us 2905.6 microseconds, and this is pretty close to 3081 microseconds! Therefore, by analysis and by code we can make the assumption that our worst case for our coloring implementation is **O(V^2 + E).**

Vertex Ordering Capabilities

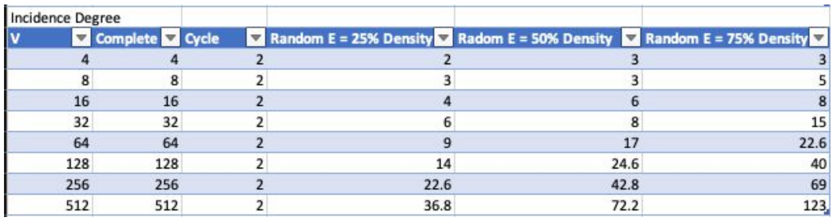
Colors Needed

For the purposes of the report we will average all of our colors over 5 times. Because of this we may have some results with decimals.



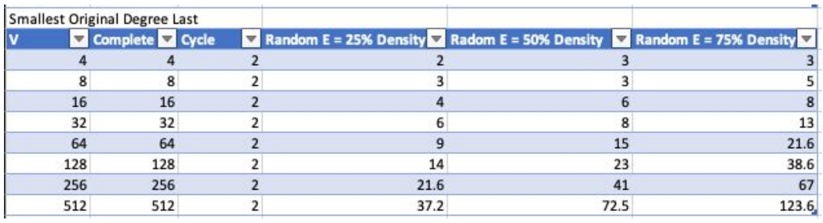
Smallest Last Vertex

V	Complete	Cycle	Random E = 25% Density	Random E = 50% Density	Random E = 75% Density
4	4	2	2	3	3
8	8	2	3	3	5
16	16	2	4	6	8
32	32	2	6	8	14
64	64	2	9	15	23
128	128	2	14.4	24	40.6
256	256	2	22.8	41.6	67.4
512	512	2	37.8	71.6	120.4



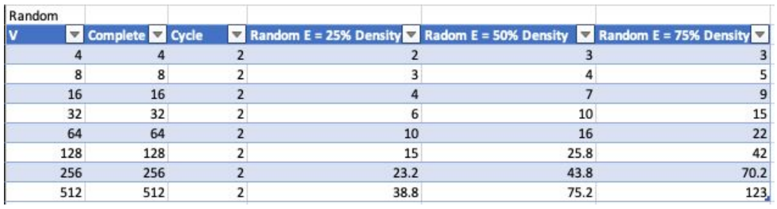
Incidence Degree

V	Complete	Cycle	Random E = 25% Density	Radom E = 50% Density	Random E = 75% Density
4	4	2	2	3	3
8	8	2	3	3	5
16	16	2	4	6	8
32	32	2	6	8	15
64	64	2	9	17	22.6
128	128	2	14	24.6	40
256	256	2	22.6	42.8	69
512	512	2	36.8	72.2	123



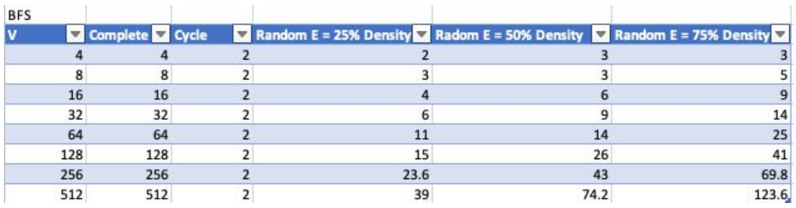
Smallest Original Degree Last

V	Complete	Cycle	Random E = 25% Density	Radom E = 50% Density	Random E = 75% Density
4	4	2	2	3	3
8	8	2	3	3	5
16	16	2	4	6	8
32	32	2	6	8	13
64	64	2	9	15	21.6
128	128	2	14	23	38.6
256	256	2	21.6	41	67
512	512	2	37.2	72.5	123.6



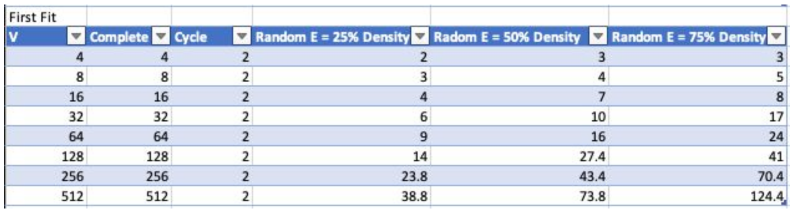
Random

V	Complete	Cycle	Random E = 25% Density	Radom E = 50% Density	Random E = 75% Density
4	4	2	2	3	3
8	8	2	3	4	5
16	16	2	4	7	9
32	32	2	6	10	15
64	64	2	10	16	22
128	128	2	15	25.8	42
256	256	2	23.2	43.8	70.2
512	512	2	38.8	75.2	123



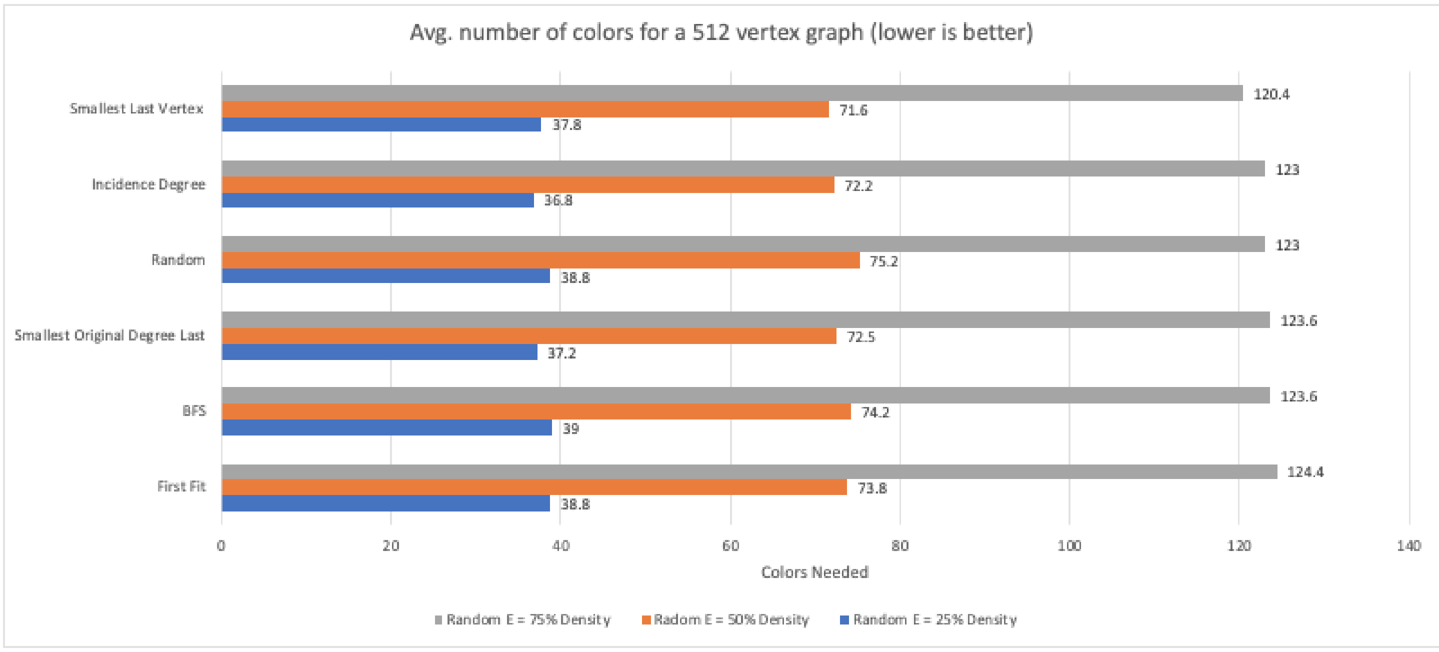
BFS

V	Complete	Cycle	Random E = 25% Density	Radom E = 50% Density	Random E = 75% Density
4	4	2	2	3	3
8	8	2	3	3	5
16	16	2	4	6	9
32	32	2	6	9	14
64	64	2	11	14	25
128	128	2	15	26	41
256	256	2	23.6	43	69.8
512	512	2	39	74.2	123.6



First Fit

V	Complete	Cycle	Random E = 25% Density	Radom E = 50% Density	Random E = 75% Density
4	4	2	2	3	3
8	8	2	3	4	5
16	16	2	4	7	8
32	32	2	6	10	17
64	64	2	9	16	24
128	128	2	14	27.4	41
256	256	2	23.8	43.4	70.4
512	512	2	38.8	73.8	124.4



Inspection of the data reveals that generally, our more computationally intense ordering algorithms produce graphs that use the fewest amount of colors, but it generally is pretty close between all of the ordering algorithms! However, smallest last vertex ordering seems to outperform *most* of the other ordering implementations in all densities, but incidence degree ordering and smallest original degree last ordering does pretty well with the 50% and 25% density graph.

According to the 25% density graph it seems like incidence degree ordering provides the fewest amount of colors on average. Incidence degree ordering provides 36.8 average colors on a random graph of 25% density. This is a 3% decrease in the total amount of colors provided by smallest last vertex (37.8) some of this can most likely be attributed to the random graphs because every iteration runs on a different random graph. However, incidence degree provides a more notable percentage increase vs. first fit and random (38.8) which equates to about a 5.5% decrease in the amount of colors used!

The 50% graph provides similar results to the 25% graph, but this time smallest last vertex is ahead of the other ordering algorithms. Smallest last vertex (71.6 average colors used) boasts only a small increase of about 1% decrease in the amount of colors used versus its next best ordering algorithm, incidence degree (72.2 colors used). However, it boasts a more significant improvement over first fit (73.8 colors used) and random (75.2 colors used) at an improvement rate of 3% and 5% respectively. Similarly to the conclusion that we drew to the 25% density we can observe that my implementations of incidence degree and smallest last vertex use the fewest number of colors.

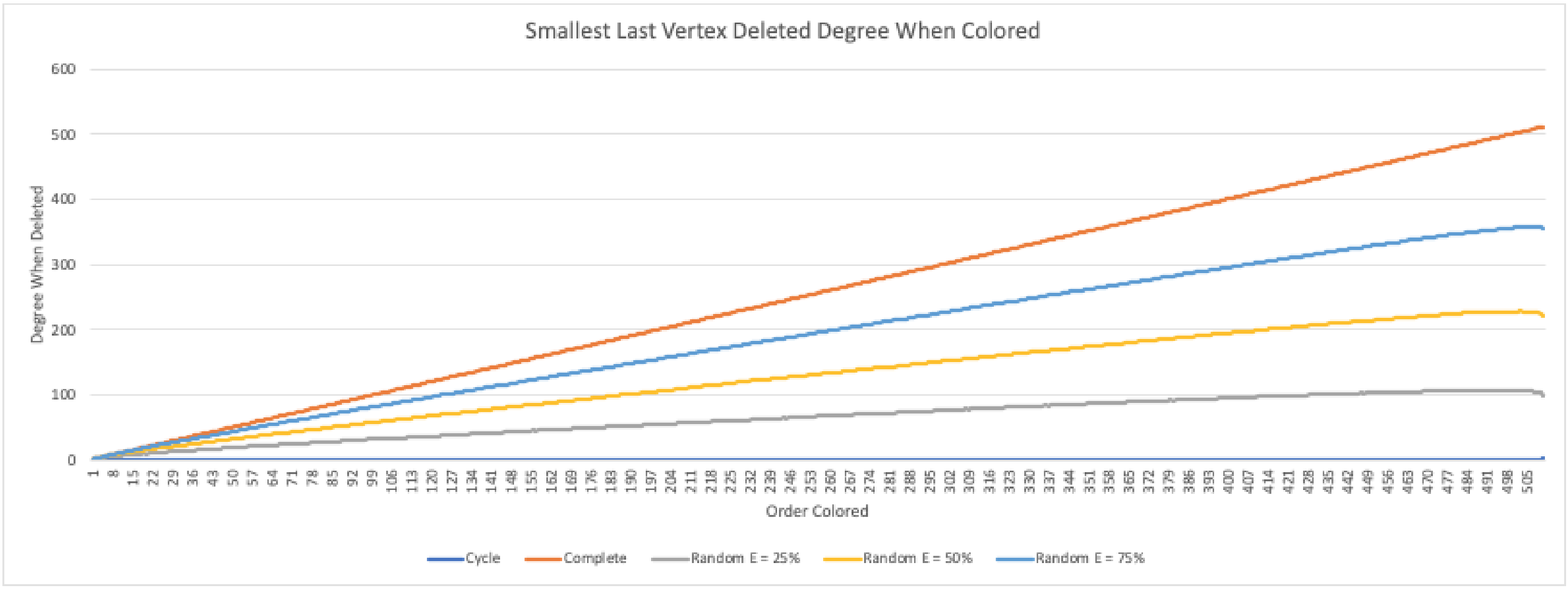
Finally, the 75% graph shows us a conclusion that we are familiar with from the 25% and 50% graphs: incidence degree ordering and smallest last vertex ordering using the fewest average colors. However, in this graph smallest last vertex performs (120.4 average colors) much better than incidence degree (123 average colors) which is also tied with random ordering (123 average colors). Smallest last vertex ordering performs about 2.15% better than incidence degree ordering, and it performs about 3.3% better than last place (first fit - 124.4 average colors).

Smallest last vertex always seems to earn 1st or 2nd place on all graph densities. Therefore, **smallest last vertex is the best ordering method in this report for minimizing the amount of colors used**. This agrees with [4] which also labels smallest last vertex as better than all of the other ordering methods I have included in this project.

It is important to weigh the different requirements when picking an ordering method. If one is willing to spend more time on ordering for achieving a decrease in the amount of colors used, then an ordering scheme like smallest last vertex or incidence degree should be prioritized. However, if speed is the most important factor, then coloring schemes that take less time like the smallest original degree last ordering and even random ordering may be considered.

Smallest Last Vertex - Revisited

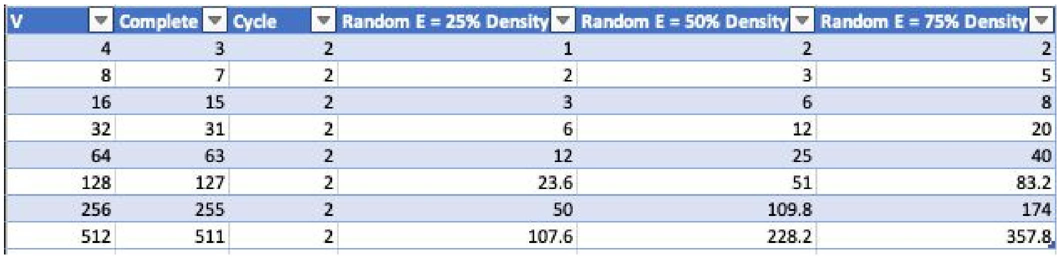
In this section we will go into detail about the coloring capabilities of the smallest last vertex ordering, the terminal cliques, and how this influences discussion on the number of colors used. Firstly we will look at the degree when each vertex is deleted.



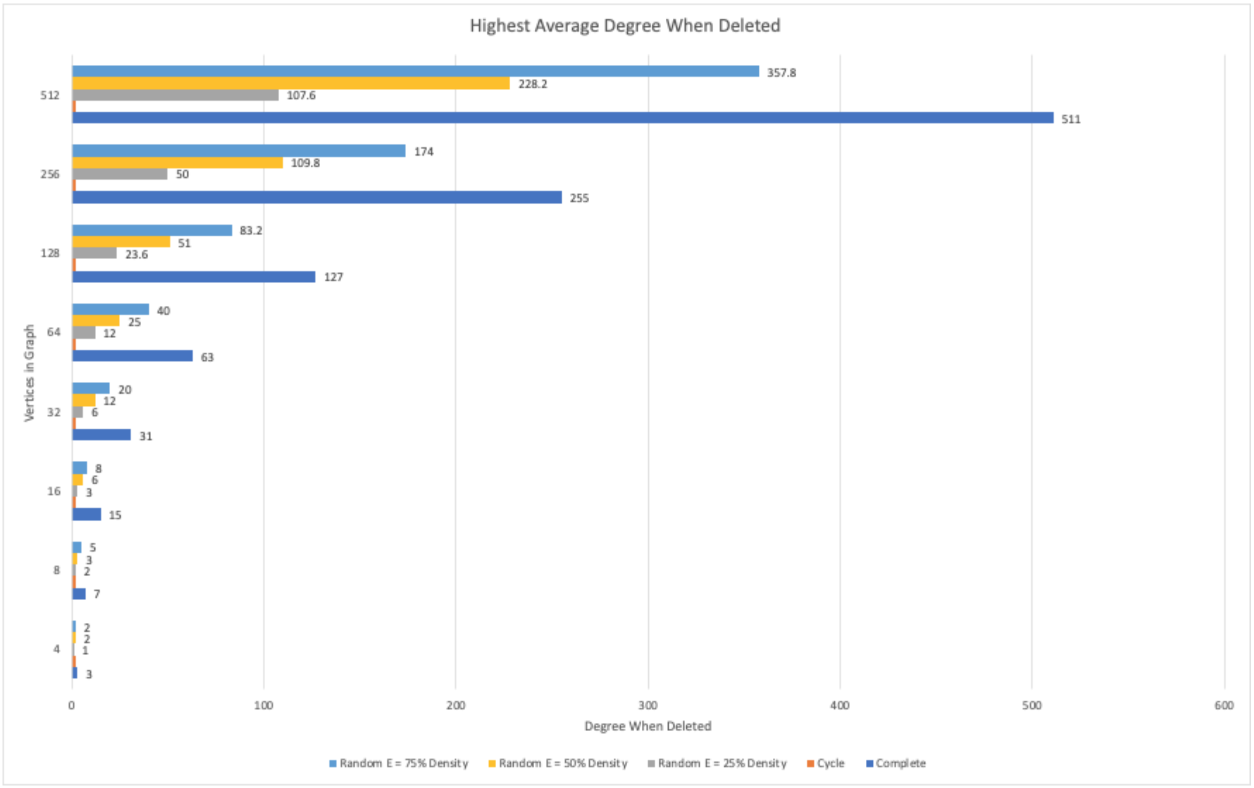
One thing to keep in mind while we are looking at this chart is that the last vertices that were colored were also the first ones deleted, so the vertices at the end of the chart were deleted first, but colored last. Something interesting to note here is that the random graphs at the end have a slight hook to them. This is because smallest last vertex finds the vertex with the lowest degree first. I assume after exhausting the lower degree vertices the algorithm eventually moves to higher degree vertices.That is why the graph seems to “hook” and then have a linear slope. However, this is not the case with the complete graph because it does not have any low degree vertices: all vertices have the same degree at the beginning.

Now I will show the average highest degrees when deleted for smallest last vertex ordering along with the average terminal clique size.

(Highest Average Degree When Deleted)

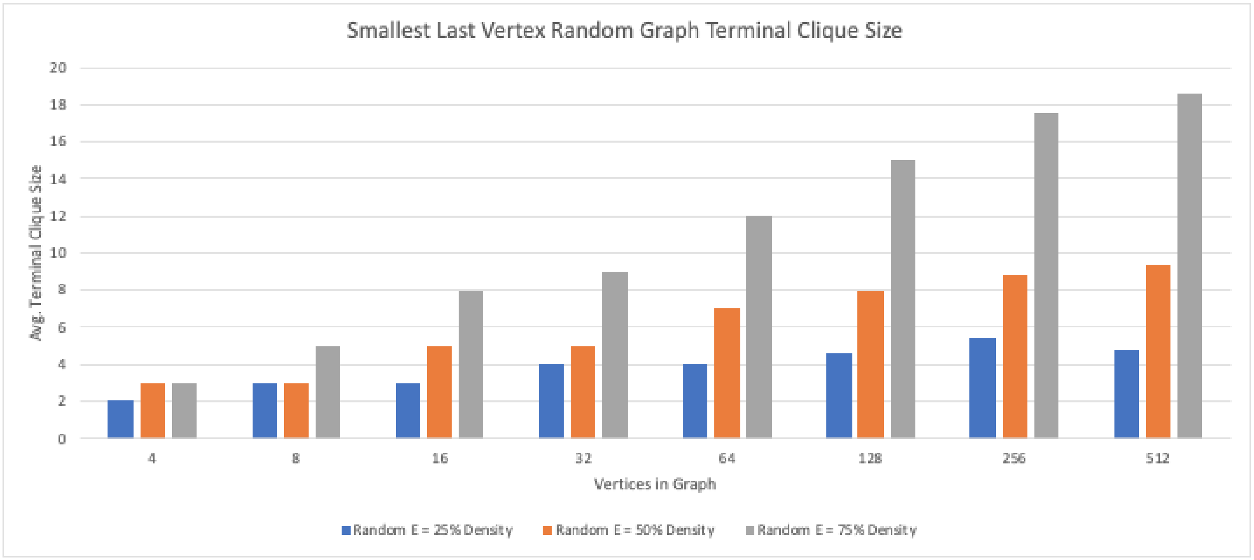


V	Complete	Cycle	Random E = 25% Density	Random E = 50% Density	Random E = 75% Density
4	3	2	1	2	2
8	7	2	2	3	5
16	15	2	3	6	8
32	31	2	6	12	20
64	63	2	12	25	40
128	127	2	23.6	51	83.2
256	255	2	50	109.8	174
512	511	2	107.6	228.2	357.8



(Average Terminal Clique Size)

V	Complete	Cycle	Random E = 25% Density	Random E = 50% Density	Random E = 50% Density2	Random E = 75% Density
4	4	2	2	3	3	3
8	8	2	3	3	3	5
16	16	2	3	5	5	8
32	32	2	4	5	5	9
64	64	2	4	7	7	12
128	128	2	4.6	8	8	15
256	256	2	5.4	8.8	8.8	17.6
512	512	2	4.8	9.4	9.4	18.6



Both the largest degree when deleted and the size of the terminal clique illustrate a bound on the number of colors in a graph. The size of the terminal clique is a lower bound on the number of colors needed while the largest degree when deleted + 1 represents the upper bound of colors needed in a graph. The range from the terminal clique and the largest degree when deleted + 1 agrees with the average number of colors needed that we discussed in Colors Needed as we never used any fewer colors than the average terminal clique. We also never used more colors than the highest degree when deleted + 1. We only reach the upper and lower bounds on our complete and cycle graphs as they provide a unique scenario, and this is not the norm.

Conclusions

Overall it is clear that smallest last vertex ordering is the best out of the other implementations to limit the amount of colors used on a graph. Going back I wish I would have spent more time lowering the upper bound on my graph generation techniques, but generating pairs of vertices can take a very long time to do multiple times for the non-uniform distributions. It is important to know one's requirements before picking a graph and its ordering method. Depending on requirements, one may opt for faster ordering methods as a trade off for more colors on a graph.

References

1. Matula, David W., and Leland L. Beck. "Smallest-last ordering and clustering and graph coloring algorithms." *Journal of the ACM (JACM)* 30.3 (1983): 417-427.
2. Coleman, Thomas F., and Jorge J. Moré. "Estimation of sparse Jacobian matrices and graph coloring blems." *SIAM journal on Numerical Analysis* 20.1 (1983): 187-209.

3. Beamer, Scott, Krste Asanovic, and David Patterson. "Direction-optimizing breadth-first search." *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012.
4. Hasenplaugh, William, et al. "Ordering heuristics for parallel graph coloring." *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*. 2014.

CODE IS ATTACHED